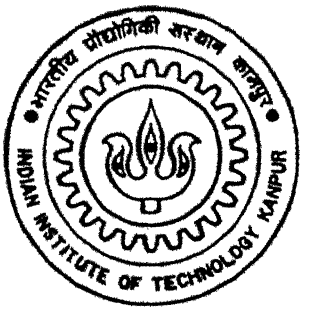
ASIC Design For NLP Applications

by

Sukumar Puvvala



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1996

ASIC Design For NLP Applications

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

M. Tech

by

Sukumar Puvvala

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April 1996

CSE-1996-M-PUV-ASIC

CERTIFICATE

It is certified that the work contained in the thesis entitled "*ASIC Design For NLP Applications*", by Mr. *Sukumar Puvvala*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. A.K.Jain)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

11/4/96

April 1996

Abstract

The field of machine–translation has matured a lot since the first generation systems which utilized the direct approach. The idea of memory based reasoning systems and example–based machine translation systems picked up in the late eighties. These systems, however, needed a very large example–base and a very powerful search engine. The HEBMT optimizes on the size of the example base by using a filtered and abstracted example base. The two bottlenecks in a HEBMT system are the dictionary search and the minimum distance calculation procedures. These algorithms implemented in hardware can significantly speed up machine translation.

In this work, we have proposed a massively parallel architecture for HEBMT and designed two ASICs - *DiSe* for dictionary search and *MiDiCal* for minimum distance calculation . The *DiSe* performs binary search on the dictionary words to find the exact word. It tries to detect a mismatch between different words in a single comparison. It also uses a pre comparison criteria to decide whether at all to compare two words. The *MiDiCal* calculates the distance between the input sentence and all sentences in the partition. Internally, it uses 4 adders, 4 dividers and 4 multipliers in parallel. It finds out the sentence in the example base which is closest in meaning to the input sentence. The *DiSe* processor and the *MiDiCal* were modeled at the behavioral level in VHDL and Verilog respectively. The models were tested using a test bench circuit.

Acknowledgements

I take this opportunity to thank my guide, Dr. A.K.Jain, who has helped me throughout this work, right from choosing NLP as the problem domain till the very end.

I would also like to thank my parents and my little sister for their constant encouragement which helped me through testing times.

The modeling of the *DiSe* processor would not have been possible without the help of Dr. Chandrashekhar and his lab personnel at CEERI Pilani. A big thanks to them for extending their lab facilities to me. Thanks are due to Mrs. Renu Jain for her initial help on HEBMT.

Thanks to all my old friends who kept my tempo going through their letters, especially Gayu and Pinky. Last but not least, thanks to all my friends here who made my stay at IIT Kanpur a *"memorable"* one.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 General Introduction

The problem of machine translation has intrigued linguists and computer scientists alike for quite some time now. Decades of research have gone in trying to find a solution to this problem and we are not even close to solving it. The reasons for it are manifold. To solve a problem fully one must first be able to comprehend it. The problem of machine translation is not even fully understood. Comprehending the problem would require a detailed knowledge of fields as diverse as linguistics, computer science, mathematics, logic, biology, information theory etc.. This list is by no means complete as new perspectives add to it. The basic question that needs to be answered is – " How do two people communicate ?". Is language the only means of communication ? Does spoken language have some absolute, fixed meaning or is it just a vehicle for conveying the meaning of some higher understanding between the two entities ? As far as human beings are concerned, sometimes a mere wink of the eye says much more than words can ever express. The term machine translation sounds like a contradiction in itself. How can a dumb, emotionless machine translate natural language ? Well, that would have been the end of the story had it not

been for the utility of machine translation. The WWW(World Wide Web) has succeeded in stringing together the scientific community of the earth. However, there is nothing it can do about the language barrier. How convenient it would be for an American scientist to understand the results published in a Japanese journal if he had a Japanese to English translator engine with him! While *human* androids may still exist in the realm of science fiction, document processing is something very much in the state–of–art. Scientists have achieved a fair amount of success in solving subsets of the problem of machine translation. The approaches have varied and matured from time to time.

A feature that most of these grand challenge problems share is that they require tremendous computing power. Most of the current implementations run on massively parallel processors. Though the computing world in general seems to be moving away from mainframes to desktops, this is one area where these large machines will be useful for a long time to come. Each one of these grand challenge problems has its own unique requirements. In general it is not possible to build a generic massively parallel system which can handle the idiosyncrasies of each problem. Building such massively parallel systems is an art in itself. When these systems run real time applications, fault tolerance becomes a crucial issue. Fault tolerance is just one of the many issues that must be kept in mind while designing these systems. In order to notch up the speed of these systems most of the functionality is committed to hardware. This is typically done by implementing critical functions in ASICs (Application Specific Integrated Circuits).

ASIC design has matured a lot now. It is no longer considered to be so arcane, though it still is an art. This has been possible because of the rapid developments in the field of EDA (Electronic Design Automation) tools. The state–of–art of these tools is still not good enough to match custom designs. However, the ASIC design cycle has been greatly simplified. Design cycle times have been greatly reduced.

With this the time to market and the cost to market have also come down. ASICs are being used increasingly in custom solutions. In this thesis we have chosen NLP (Natural Language Processing) as the problem. We have identified some functions of the system which if implemented in hardware can significantly speed up machine translation. We have tried to provide ASIC solutions to these functions.

1.2 Literature Survey

The first generation machine translation systems utilized a naive, direct approach. The second generation systems used a rule based transfer approach [14]. A new generation began when the emphasis shifted from linguistic considerations to exploiting language based on corpus, statistics, and examples [10]. Example based−approach was first proposed by analogy by Nagao[11]. The same idea was then presented as memory−based reasoning by Stanfill and Waltz [15], and case−based reasoning by Reisbech and Schank[12]. Sumita and Iida proposed an EBMT (Example Based Machine Translation) system. Sato and Nagao [13], highlight on the problem of utilizing more than one example for translating more than one sentence.

Kitano [5], has presented an architecture for MBMT (memory−based machine translation) system. In MBMT system, translation is attempted from the raw examples by invoking the best suited example using a distance measure. Kitano et. al. [5], implemented an EBMT system on an IXM−2 massively parallel associative memory processor and a CM−2 Connection Machine. Lewis Tucker surveys some massively parallel architectures in [8].

1.3 Aim of the Thesis

Kitano et. al.[5], have proposed the use of massively parallel processors for solving a number of grand challenge problems like EBMT and chess. Brute force algorithms

can never be a substitute for good algorithms. On the flip side of the coin, brute force coupled with good algorithms can surely work wonders. The state–of–art in algorithms and technology advocates the use of massively parallel architectures to solve complex problems.

The aim of this thesis is to propose a massively parallel implementation of the HEBMT (Hybrid Example Based Machine Translation) architecture and model two ASICs which can be used in the proposed architecture. The first of the ASICs is a dictionary search processor, an important step in any machine translation system. The second ASIC is used to find the example which has the minimum distance from the source language input sentence.

1.4 Layout of the Thesis

The thesis report is arranged as follows

1. Chapter 2 discusses the design of a hypothetical massively parallel architecture
2. Chapter 3 discusses the design of the *DiSe* processor
3. Chapter 4 lists the simulation results of the *DiSe* processor
4. Chapter 5 discusses the design and simulation results of the *MiDi* processor
5. Appendix A contains the component definitions of the *DiSe* processor in VHDL
6. Appendix B contains the component definitions of the *MiDi* processor in VHDL

Chapter 2

DESIGN OF A MASSIVELY PARALLEL ARCHITECTURE FOR MACHINE TRANSLATION

2.1 Introduction

In this thesis we present the design of a Massively Parallel Architecture for Machine Translation using the hybrid example based approach for machine translation (HEBMT) by Jain et. al.[1]. The HEBMT approach integrates the strategies of a rule/pattern based approach and the example based approach. The main problem with any example–based system is the efficient processing of a large example base which requires a large "memory" and tremendous processing power, beyond the capacity of ordinary sequential computers. These systems, however, lend themselves to tremendous data parallelism. Therefore we have selected this approach for designing the parallel architecture.

The functional diagram of HEBMT is shown in Figure 2.1. The basic approach to HEBMT can be summarized as follows : " Given an input source language (SL)

sentence , find an example, from the example base, closest in meaning to the input sentence and perform some word level transformations to get the translation."

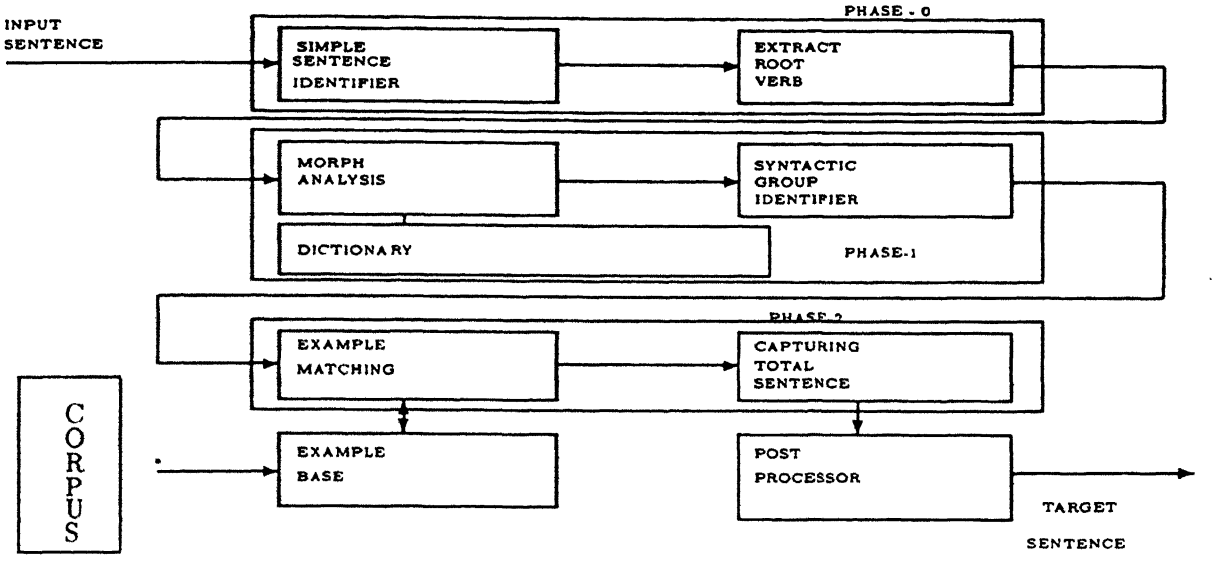


Figure 2.1: Block diagram of a HEBMT system

The example base assumes gigantic proportions for large practical systems. The HEBMT optimizes on space by storing abstracted examples rather than raw examples. Raw examples are the complete sentences in natural language. An abstracted example is collection of syntactic units, where each syntactic unit is a group of words or a single word, such that there is a tight binding between the words of a syntactic unit. Translation time largely depends on the size of the example base, hence both the space required to store example base and the time for finding the best match, are reduced in HEBMT. By devising suitable data structures and efficiently partitioning the dictionary and example base, the search can be carried out in parallel by a large

number of processors, reducing processing time. In the thesis we have also discussed such partitioning schemes.

In order to facilitate speedy translation, the process is divided into three phases

1. Phase−0 involves splitting the input sentence into simple sentences and identification of the root verb.

2. Phase−1 involves morphological analysis and identification of the syntactic group of the sentence.

3. In phase−2 the distance of the input sentence from every other example stored in a partition of the example base is calculated. The example with minimum distance is the translation.

The basic operation in morphological analysis is the dictionary search, which can be carried out in parallel. There are basically two ways in which the search can proceed viz. :

1. Simple−search and

2. Expectation−driven approach

2.2 The Simple−Search Approach

In this approach the morphological analysis and the syntactic category identification are carried out sequentially in that order. The search procedure handles only the morphological analysis. The yield is then fed to a finite−state machine which identifies the syntactic group of a sentence.

The data−dictionary is organized with the words serving as keys. The words are arranged in the same format as one would find in the Oxford dictionary. Even in the simple search approach the search can be performed either for one word at a time or for several words in a sentence at the same time.

2.2.1 One Word at a Time

When an input sentence arrives the first word is searched for in the dictionary. After a successful search the second word is searched and so on for all the words in the sentence. At any time, only one word is being searched in the dictionary. The advantage of this method is that it is very simple and straightforward.

2.2.2 Several Words at the Same Time

In this method, the search for several words in the input sentence proceeds at the same time, in parallel. If the size of the partition in which a word is to be searched is large then a greater number of processors can be assigned to look for the word. This approach leads to better throughput than the previous approach in terms of the number of successful searches per unit time. However, it is also more complicated. The scheduler must decide on the number of processors to allocate for each word. Other issues like uniform load balancing are also important.

2.3 Expectation Driven Approach

In this approach both the morphological analysis phase as well as the syntactic group identification phase have been integrated. In fact the morphological analysis or dictionary search is driven by the syntactic group identification analysis henceforth referred to as ISG. The motivation behind this approach is to simulate the working of the human brain, in a similar scenario.

The first implication of this approach is reflected in the organization of the data–dictionary. The words no longer serve as the keys. At a very broad level the disk is partitioned on the basis of the syntactic categories of the words. For example all nouns go in one partition , all pronouns in another partition and so on. A second level index based on the alphabets in the word is used to reduce the search space.

The ISG is a finite state machine which captures the syntactic group of a sentence. A part of the ISG is shown in Figure 2.2. An example will make things more clear. Let the input sentence be :

"MARY HAD A LITTLE LAMB ".

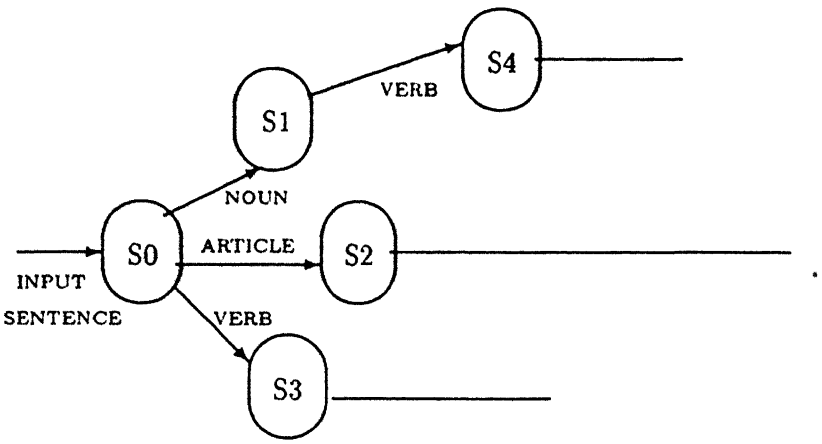


Figure 2.2: Syntactic group identifier

In state s0 the ISG is expecting a NOUN, VERB or an ARTICLE denoted by the three transitions. Therefore the current expectations are for a NOUN, VERB or an ARTICLE. So the search procedure is directed to look for the word "MARY" in the partitions reserved for NOUN,VERB and ARTICLE. The processor which detects the word answers to the ISG in the affirmative. Note that it is possible for a word to belong to more than one syntactic group in which case more than one processor will answer in the affirmative. In such cases there cannot be a unique transition. The ISG can have more than one current state now. All transitions out of the current states will become the current expectations. The search procedure is informed of the current expectations by the ISG and the second word is dispatched. Depending on the results new expectations are generated and some older ones are eliminated. This process is repeated for all words in the input sentence. Ultimately,

after all the words in a sentence have been processed the ISG will have traced a unique path from its initial state to some final state. In the process, it identifies the syntactic group of the input sentence, which in this case happens to be "$< np > < vp > < np >$", where np stands for noun phrase and vp stands for verb phrase.

The above technique can be improved by avoiding the traversal of unnecessary transitions. This technique simply uses a lookahead of one. Consider the same input sentence as above. The words "MARY" and "HAD" along with their expectations are passed on to the search procedure. The search proceeds exactly as described above. Once the syntactic category of the second word is known, in most cases, it will help to eliminate some unnecessary transitions. This approach can be carried on further but may complicate the scheduler which allocates search processors to words.

The expectation driven approach is expected to be faster than the simple search approach. The reason for this optimism is that the search space is reduced in the former case. For example, the search space for an input word "wince" will be the set of all verbs beginning with 'w' in the expectation–driven approach as against the set of all words beginning with 'w' in the simple–search approach. Obviously the search space is smaller in the first approach.

2.4 The Parallel Architecture

The proposed parallel architecture is shown in Figure 2.3. The host is a front end computer which accepts source language sentences from the outside world. These sentences after some processing are dispatched to the phase–1 master controller. The master–controller connects to a cluster of simple processors (SPs) through a common bus. The operations are pipelined in three stages where each stage corresponds to each phase of the HEBMT.

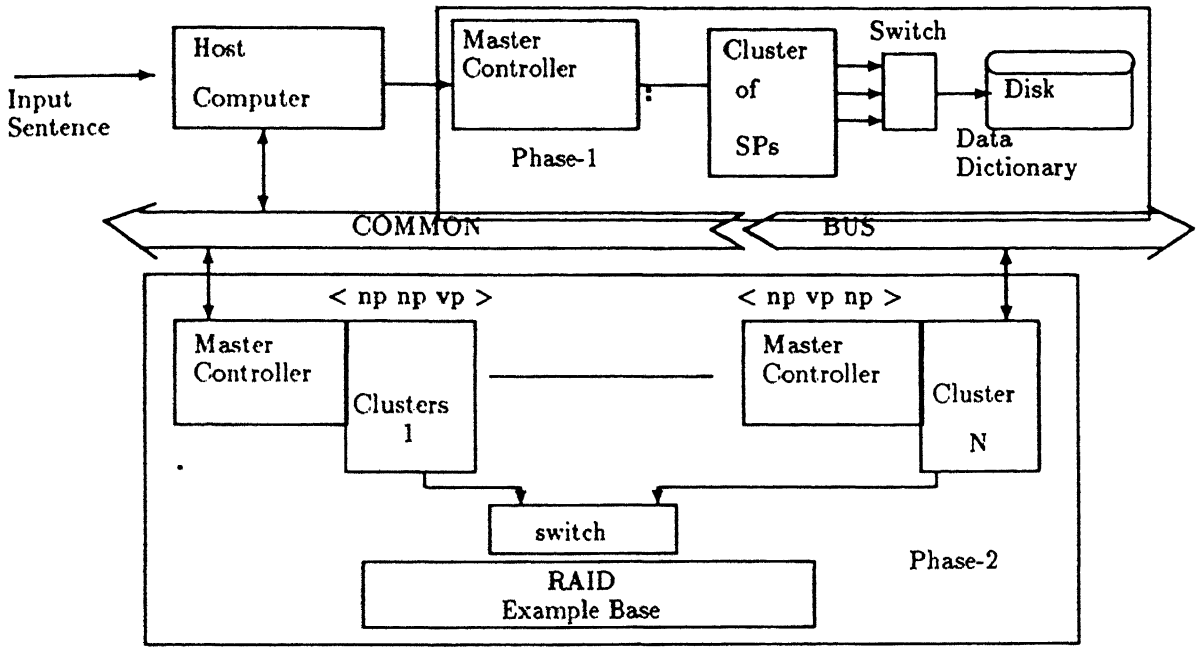


Figure 2.3: Overall system architecture

In Figure 2.4 the SPs have a local/shared 4k RAM. Each local RAM is shared by an SP and the master–controller. The master–controller loads data from the disk into the local RAM of each SP and fires the search procedure. All disk traffic is routed through the common high bandwidth bus. Each SP accesses only its local memory. The main advantage of this method is its simplicity.

An improvement over the above architecture is shown in Figure 2.5 where each SP has a local/private cache in addition to a local/shared RAM. The memories are "active" memories(Asthana et. al.[3]). This enables the SPs to schedule their own disk operations through dedicated channels. Thus most of the disk traffic is kept off the common bus. This leads to lesser communication contention on the common bus as compared to the organization described in the previous paragraph. Since each SP accesses only its local/private cache and local/shared RAM there is no memory contention. All the clusters have access to a single multi–port disk. Alternatively

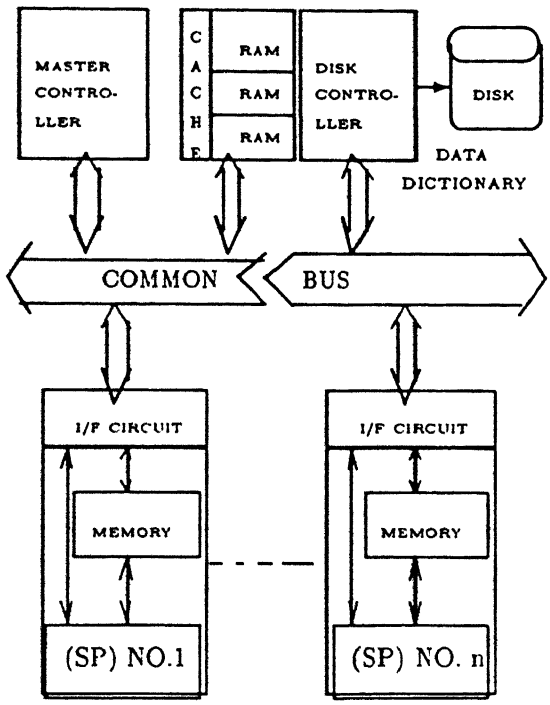


Figure 2.4: Phase–1 system architecture

there could be many disk volumes serving groups of clusters. Data is replicated in different disks for fault–tolerance.

It is worth mentioning that cache coherence is not a problem here. This is because no two simple processors operate on the same data at the same time and no two caches have a copy of the same data at the same time. Thus the overheads involved in maintaining cache coherence, bus–snooping, invalidate–on–write, etc., are eliminated. Each SP has access only to its local cache and a RAM which it shares with the master controller. The sharing takes place on a strict master slave basis so memory contention is eliminated. However some other fault–tolerance issues are involved. What happens if an SP fails? All the data in its cache/RAM will go unchecked. In the worst case the data being searched may be in the failed processor's cache/RAM. A simple but expensive solution is to employ hardware redundancy

techniques and use a backup processor for every SP. The master controller must have a way of detecting the failure of an SP and then transferring control to the backup processor. Alternatively the backup processor polls its primary SP and take over in the case of a failure.

A significant performance gain is achieved by caching the results of searches. The caching can be done only at the master controller but the cache size will be large. Another approach is to use two level caching, one at the master controller and a second level at the SPs. The search procedure is a bit modified now. Given an input stream the dictionary search for the first few sentences proceeds just as outlined above except that the results of searches are simultaneously cached. For subsequent sentences most of the search requests can be satisfied from the cache itself. This is because only a small fraction of the words in a natural language have a high frequency of occurance.

Disk I/O latency is further masked in the system shown in Figure 2.5 When an SP receives a word from the master controller it immediately begins the search in its cache while simultaneously scheduling a disk i/o. The master controller maintains a data structure SPMCT (SP memory content table) to keep track of data stored in the memory M_i of SP_i. Before dispatching a word to an SP, it checks if any SP already has the disk block in its memory. The word to be searched can then be directly sent to that particular SP. This way a lot of unnecessary disk i/o can be eliminated. Similarly it is also possible to pre-fetch data thus overlapping search with i/o. Consider the scenario where the master controller has just dispatched a sentence S_n for processing to the SPs. Now, instead of waiting for the results of S_n it can proceed and fetch the next input sentence, S_{n+1}. The words in S_{n+1} are passed through a filter which eliminates all those words whose information is already in the cache or memory. For those words which warrant a disk i/o , the master controller promptly issues a disk i/o request. A part of the RAM serves as a buffer pool.

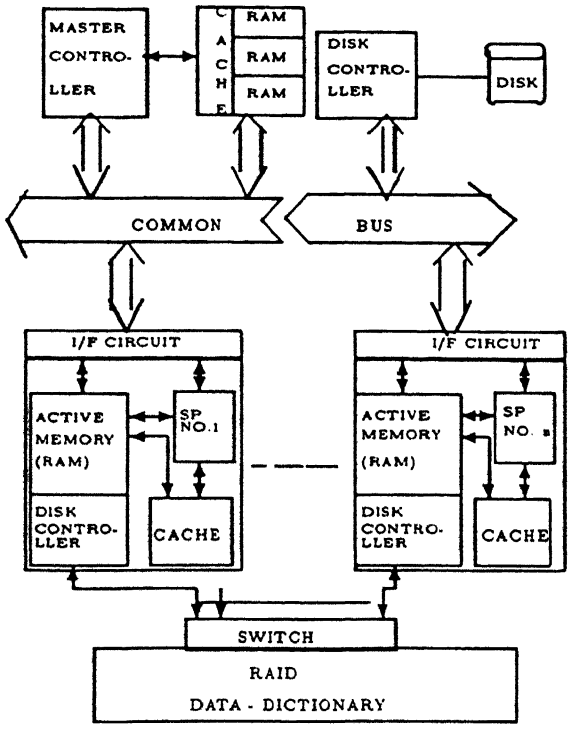


Figure 2.5: Improved phase−1 system architecture

Thus i/o for sentence S_{n+1} is overlapped with the search for S_n. Eventually when the processing of sentence S_{n+1} starts the data requests can be satisfied from the buffer pool itself. This is a simple memory to memory copy which is many orders of magnitude faster than a disk to memory copy.

It may be pointed out that the basic hardware in both the cases, i.e, simple−search and expectation−driven approaches, is the same.

PHASE−2

The host computer dispatches the sentence along with all the information gleaned in phase−1 to the appropriate cluster in phase−2. It consists of a series of disjoint clusters, each dedicated to a particular syntactic group. All the clusters are connected via a common bus to The host computer. Each cluster consists of many simple processors. Each SP has a local cache and RAM which is shared by the

controller of that cluster. Having a single cluster for a syntactic group introduces a single point failure for all sentences belonging to that syntactic group. So for fault–tolerance reasons we let every syntactic group to be handled by two clusters. For example, the first cluster may handle all sentences of type "$< np > < np > < vp >$" and "$< np > < vp > < np >$ ". The second cluster may handle "$< vp > < np >$" and "$< np > < vp > < np >$" and so on.

There are certain design issues which are discussed next. Firstly the issue of uniform load– sharing among the clusters must be handled. The tradeoffs of having disjoint clusters as against a common pool of processors is to be considered. All The processors in this pool could work on a single sentence in parallel. Thus the stream of input sentences would have to be processed sequentially. However, the probability of two successive sentences, in an input stream, belonging to the same syntactic category is remote. Making use of this property it is possible to dispatch different sentences to different clusters. Thus we can have different clusters operating on different sentences at the same time. This partially "solves" the problem of load–sharing.

The example base is partitioned on the basis of the syntactic categories of sentences. All sentences of type "<np> <np> <vp>" go in one partition. In another partition we have sentences of type "<np> <np> <vp>" and so on. This helps in narrowing down the search space. A second level of partition is based on some other attributes of the sentence. We have physically separate disk volumes storing one or more partitions in a each disk. Data is also replicated on separate disks. This introduces multi-point failure. Each disk is dedicated to one or more clusters. This design , to an extent, mitigates the i/o bottlenecks as compared to the design where all the clusters access the same disk.

A sentence, besides belonging to a particular syntactic category can have many attributes. In fact each phrase in a sentence is tagged with certain attributes like

NUMBER(singular,plural), PERSON(first,second,third), GENDER etc. Depending on the attributes some weights are attached to the phrases. The distance of an input sentence from every example in a partition of the example base is calculated. This is actually a phrase by phrase comparison of the attributes. The sentence with the minimum distance is returned as the translation.

The main bottlenecks in the architecture described above are the dictionary search procedure and the distance calculation procedure. The following chapters discuss the design of two ASICs. Chapter 2 discusses the design of the *DiSe* processor which is a hardware implementation of the dictionary search algorithm. Chapter 4 discusses the *DiSe* simulation results. Chapter 5 describes the design of the MiDiCal processor which is a hardware implementation of the distance matching algorithm. The simulation results are also discussed in the same chapter.

Chapter 3

THE *DiSe* PROCESSOR

DiSe is an acronym for dictionary search. The *DiSe* processor searches a dictionary for the presence or absence of a particular input word. The "word" here means a word in a natural language. The *DiSe* is an ASIC which can be used in a machine translation system such as the one discussed in the previous chapter.

This chapter contains a detailed description of the design of the *DiSe* processor as well as the factors which contributed to the evolution of the *DiSe* processor. The *DiSe* was modeled in VHDL (Very High Speed Integrated Circuits Hardware Description Language). The *DiSe* was modeled at the behavioral level. A test-bench model, consisting of a memory-chip, a clock generator and the *DiSe*, was written in VHDL. The model was simulated with test 'programs' (actually data suites, as will become clear in the following pages). The *DiSe* is then broken down into its constituent components at the Register-Transfer level. All the components are documented in VHDL. A structural description of the *DiSe* using the components is then proposed. The outline of the chapter is as follows

- Introduction.
- Problem definition.

- How to solve it?
- Instruction vs instructionless processor.
- Disk data format.
- Input Data format for the *DiSe* processor.
- The *DiSe* processor interface.
- Data and address bus protocols.
- Behavioral description.
- Description of the logic controller.
- Register-transfer architecture.
- Future directions.
- Pseudo-code for the *DiSe* model.

3.1 Introduction

The working of a hybrid example based machine translation system was discussed in chapter 2. The main bottlenecks in the system are

1. Dictionary searching process and
2. The distance calculation process.

The *DiSe* processor was designed to tackle the first bottleneck without any major intervention on the part of the master processor. The idea was to design a very simple, cost-effective, efficient processor which could do most of the dictionary searching work independently. So the *DiSe* doesn't have any complex architectural

features like branch prediction, out of order execution ,etc.. The primary idea was to exploit the parallelism inherent in the application. It was intended that a large number of such processors could be used in a massively parallel environment so that the dictionary search process could be handled by a "package" of n such processors.

Emerging technologies like WSI - wafer scale integration- and multi-chip modules would make it possible to pack hundreds of such chips on a single silicon wafer. Having a *DiSe* processor and its associated memory on the same chip would significantly speed up the *DiSe*.

3.2 Problem Definition

The problem for which the *DiSe* has been designed , is formally stated as follows :

> " Given a set of words W and an input word(string), IS, find out if IS is a member of the set W."

$$F(W, IS) = 1 \; if \; IS \in \{W\}$$
$$= 0 \; if \; IS \notin \{W\} \tag{3.1}$$

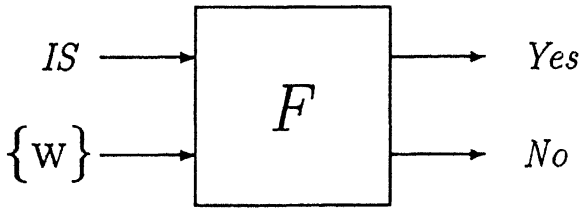


Figure 3.1: The *DiSe* concept

where F is the characteristic function defined on set W.

The set W in equation 3.1 represents a dictionary and the string IS represents the word which is to be searched in the dictionary. The *DiSe* processor is a VLSI implementation of the characteristic function F.

3.3 How to Solve It?

The section departs from the precise but abstract problem definition of the previous section to more concrete implementation aspects. First of all we need an efficient algorithm to implement F. This brings us to the first and the most important decision regarding the *DiSe* processor :

" Should it be implemented as an instruction-set processor or an instructionless processor?"

3.3.1 Instruction-set vs Instructionless Processor

There are some important factors which govern the choice of of an instruction-set processor as against an instruction-less processor. A very important factor which advocates the use of an instruction-set processor is **flexibility**. This is because an instruction set processor can be programmed. With a judicious choice/design of the instruction set, the processor can be programmed to handle a wide variety of input data formats. The input data string, IS, and the data dictionary can be in almost any format. However, flexibility doesn't come for free. Implementation of an instruction-set through firmware, hardwired connections or nanoprogramming complicates chip design. It also occupies a lot of space on the silicon wafer which can be utilized for other purposes. Moreover the idea was not to build a complicated, generic processor but a simple ASIC.

The alternative to an instruction set processor is to go in for an instructionless processor. The disadvantage is that it can be designed to accept only a few input

data formats. The more general we try to make it the more complicated it becomes. However, as will be pointed out shortly, the scenario is not really that bad. The biggest gain comes in terms of simplicity and speed. The logic is fairly simple. For a particular input data format, the instruction set processor has to

1. Fetch the instruction.
2. Decode the instruction.
3. Fetch the data on which the instruction will act.
4. Act on the data.
5. Cleanup(reset write-back flags, flush history buffers, etc).

On the other hand an instructionless processor has to

1. Fetch the data.
2. Act on the data.
3. Possibly cleanup.

The instructionless processor has to perform two to three lesser operations as compared to an instruction-set processor. That is a lot of saving even if we consider the performance on translating the smallest of documents. Of course, by using some advanced architectural features it may be possible to mask the extra memory access latency of instruction set processors but it complicates the chip. Fortunately for our application the fixed input format is not a handicap. It is, in fact, our *strongest* motivation to go in for an instructionless processor.

It is worth pointing out here what could have have been ideal implementation of the *DiSe* processor. Around the late 80s, a technology known as the WISC - Written Instruction Set Processor - had shown great promise. A WISC processor

has a programmable firmware control store in its core. So, for any input data format, it is possible to just change the firmware and "teach" the processor, so that it doesn't spend time reading in the same instructions again and again.

3.4 Disk Data Format

The data-dictionary organization has already been discussed in section. In this section we focus on the format in which each word is stored on the disk.

The data stored in a dictionary basically consists of words. Each word has

1. A meaning or meanings and
2. Some other information such as whether it is a noun, a pronoun, a verb, masculine, feminine, singular, plural, etc..

We club the meaning of the word and all other information about it into a single entity known as the attributes of the word. Therefore each record in the dictionary contains

1. A word and
2. The word's attributes.

The C language structure definition of a data-dictionary record is as follows :

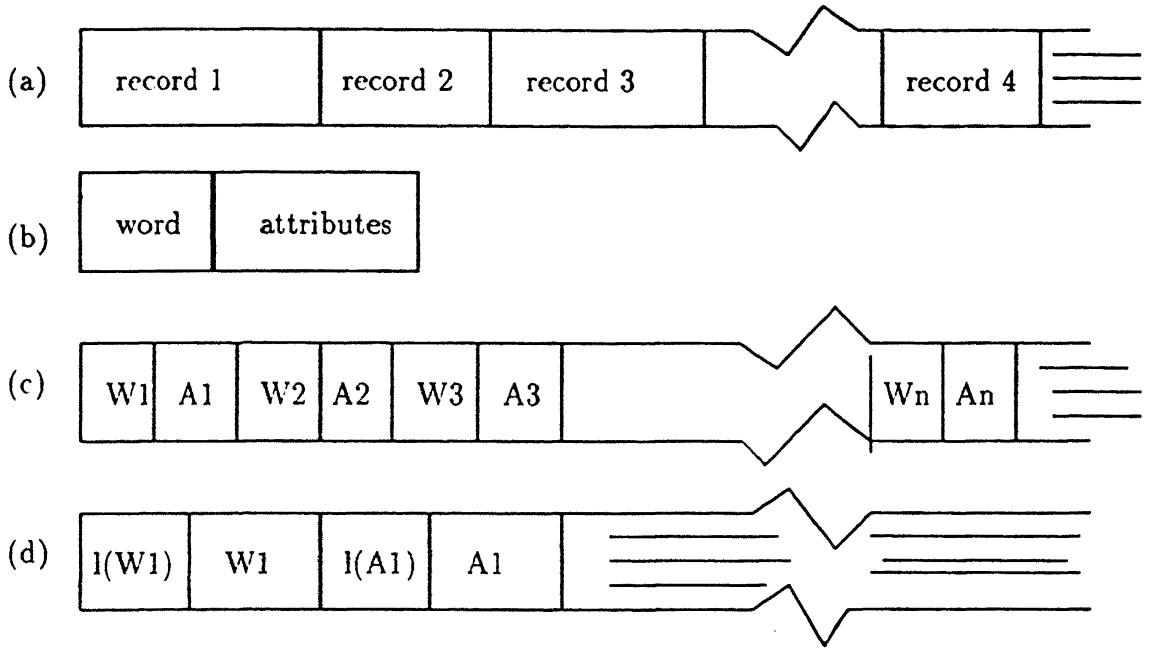
```
struct dictionary_entry {
        string word;
        src_attributes attrib;
};
```

A dictionary has a structure. This means that that there are not many new words being added to the language or deleted from it very often nor do the meanings and attributes of words change frequently.

The way each word and its attributes are stored on disk is shown in Figure 3.2. As can be seen from Figure 3.2 each word and its attributes are of different sizes. So there must be some way to know the starting and ending locations of each word. Similarly there must be some such delimiter for a word's attributes. A simple delimiter is the length of each word and its attributes. We could also have used some particular bit string or special character for delimiting word boundaries but using lengths as delimiters has tremendous advantages which will become apparent when we discuss the behavioral modeling of the *DiSe* processor. Moreover choosing a special character which is *special* to all languages is difficult. Figure 3.2(d) shows the organization with word delimiters.

The machine translation system architects are free to choose any organization for the attributes field. The overhead incurred in terms of space per word for the organization using word lengths as delimiters, is just two bytes per word-attribute pair(record). This organization is fairly *generic* in the sense that it offers the s/w writer total freedom in organizing the layout of attributes. Moreover the dictionary writer (data entry person) need not bother himself with the lengths of words and their attributes. A simple filter can do the job of calculating the lengths and inserting them into the dictionary records. There are no insertion deletion problems like internal fragmentation.

It may be pointed out that this particular organization of data has no bearing on the overall organization of the dictionary. The dictionary can be partitioned on any basis which facilitates speedy searching. It is only the leaf nodes in the search path which contain data in this format.



Wn word no. n
An attribute of word no. n
l(W1) length of word no. 1
l(A1) length of attribute of word no. 1

Figure 3.2: Data dictionary record organization

3.5 Input Data Format for the *DiSe* Processor

The data loaded into main memory can be viewed as a string of bytes. The input string can be viewed as a pattern **S**. The overall problem is then simply matching the pattern **S** in the byte string and returning the attributes for a pattern match. This is a restricted form of the more general pattern matching problem in that we look for a pattern only at a word boundary.

The main memory encapsulates all the data that have to be served as input to the *DiSe*. The input data includes

1. The pattern (word) to be searched and

2. A list of records from the dictionary which are most likely to contain the input pattern (word).

Figure 3.3 shows the main memory organization. This is the format in which data must be fed to the *DiSe* processor.

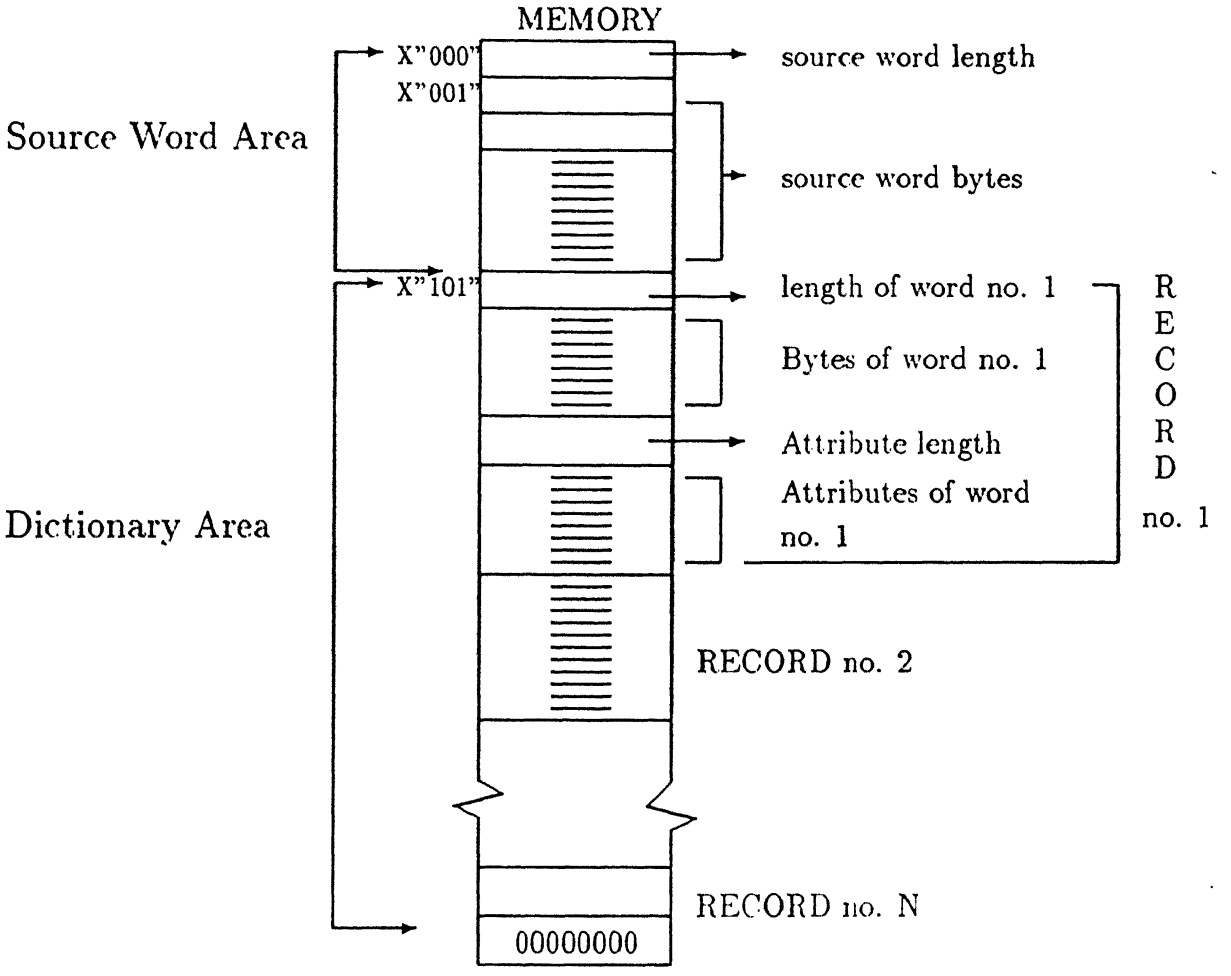


Figure 3.3: Input data format

So far we have described the inputs and the format in which they are presented to the *DiSe*. Now we focus on the *DiSe* processor.

3.6 The *DiSe* Processor Interface

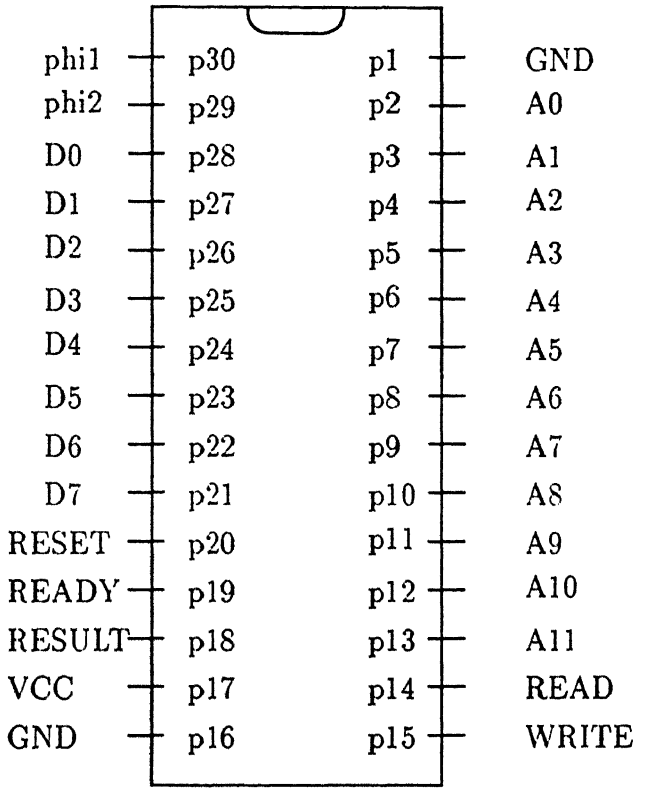


Figure 3.4: *DiSe* pin diagram

The pin diagram of the *DiSe* processor is shown in Figure 3.4. It has 27 pins of which

1. 12 pins interface to the memory address bus.

2. 8 pins interface to the memory data bus.

3. 2 pins interface to the clock generator.

4. 4 pins are for control.

5. 1 pin for result.

3.6.1 Clock Inputs

Pins P30 (phi1) and P29 (phi2) are the two clock inputs. Together, they provide a two phase non-overlapping clock for the *DiSe* processor. The clock waveforms are shown in Figure 3.5. Each cycle of the phi1 clock defines a bus_state, one of Ti (idle), T1 or T2. Bus transactions consist of a T1 state followed by one or more T2 states, with Ti states between transactions.

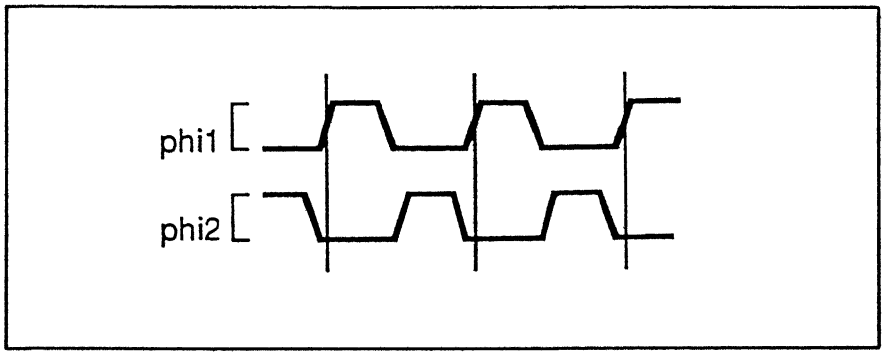


Figure 3.5: Clock waveforms

3.6.2 Address and Data Bus

The *DiSe* processor communicates with its memory over synchronous 12-bit address and 8-bit data buses. Pins P2 to P13 form the address bus while pins P21 to P28 form the data bus. The port A_bus is a 12-bit address bus. The port D_bus is a 8−bit bidirectional data bus.

3.6.3 Control Ports

The Pins P14 and P15 are the read and write ports. They control bus read and write transactions. Pin P19 stands for the ready port. This port is used by a memory device to indicate that a read data is available on the bus and or a write data has been accepted. Pin P20 is the reset port. This port is used by an external device such as the master processor to reset the *DiSe* processor. It is also known as the firing signal which sets off the *DiSe* to start its search procedure.

3.6.4 Result Port

This pin (P18) is used to announce the result of a search to an external entity. Actually it just interrupts the master processor saying that a search has ended. It does not indicate whether the search was successful or unsuccessful. The master processor has to verify the results in some other way which will be described shortly.

3.7 Bus Protocol

This section describes the *DiSe*-memory read and write protocols. The first subsection deals with the memory read protocol and the second section deals with the memory write protocol.

3.7.1 Read Protocol

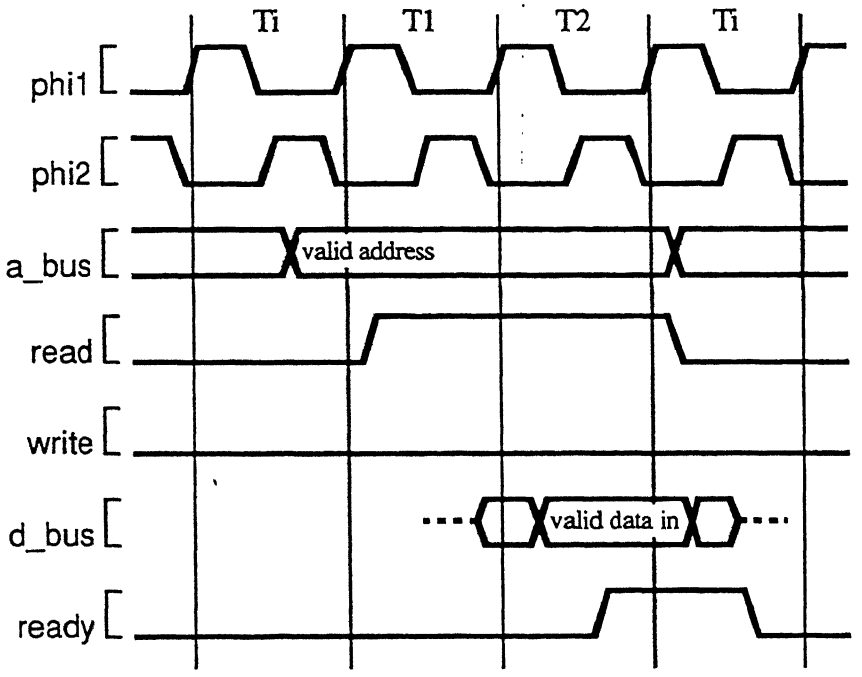
The timing for a bus read transaction is shown in Figure 3.6(a). During an idle state, Ti, the processor places the memory address on the address bus to start the transaction. The next state is the T1 state. After the leading edge of the phi1 clock, the processor asserts the read control signal, indicating that the address is valid and the memory should start the read transaction. It always leaves the write

signal negated during read transactions. During the T1 state and the following T2 state, the memory accesses the requested data, and places it on the data bus. If it has completed the data access by the end of the T2 state, it asserts the *ready* signal. The processor accepts the data, and completes the transaction. On the other hand, if memory has not yet supplied data by the end of T2 state, it leaves ready false. The *DiSe* then repeats T2 states until it detects ready true. In this way, a slow memory can extend the transaction until it has read the data. At the end of the transaction, the processor returns its control outputs to the default values, and the memory negates ready and removes the data from the data bus. The processor continues with idle states until the next transaction is required.

3.7.2 Write Protocol

The timing for a bus write transaction is shown in Figure 3.6(b). In this case also, the transaction starts with the *DiSe* placing the address on the address bus during a Ti state. After the leading edge of phi1 during the subsequent T1 state, the *DiSe* asserts the write signal. The read signal remains false for the whole transaction. During the T1 state, the processor also makes the data to be written available on the data bus. The memory can accept this data during the T1 and subsequent T2 states. If it has completed the write by the end of the T2 state it asserts the ready signal. The processor then completes the transaction and continues with Ti states, and the memory removes the data from the data bus and negates ready. If the memory is not able to complete the write by the end of the T2 state, it leaves ready false. The processor will then repeat T2 states until it detects ready true.

bus read transaction.



bus write transaction.

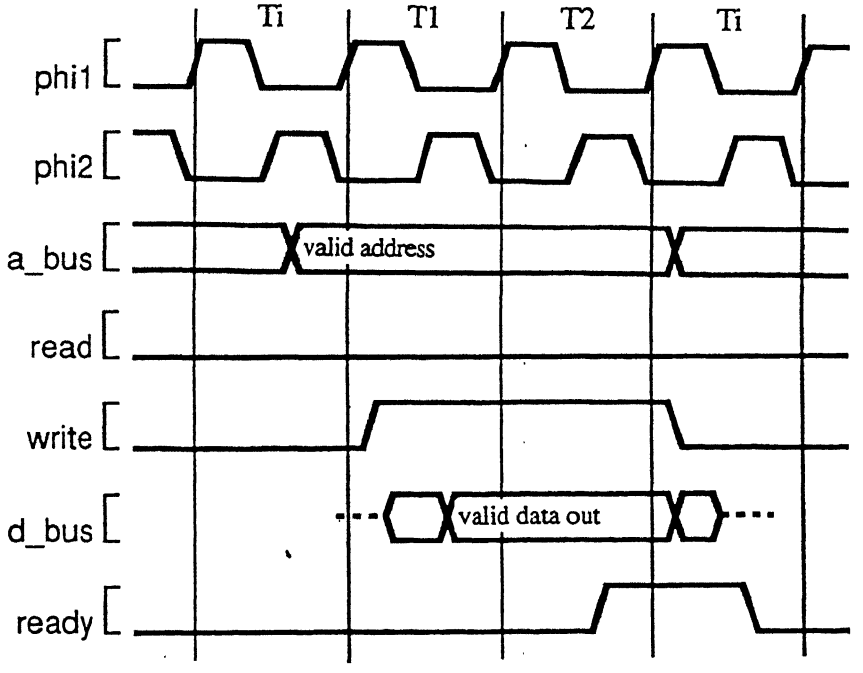


Figure 3.6: *DiSe* bus protocol

3.8 Behavioral Model

In this section a behavioral description of the *DiSe* processor is presented. This model is used to run test programs (actually data suites) by connecting it to a simulated memory model.

We begin the behavioral description with an entity declaration of the *DiSe* processor shown below :

```
entity DiSe is
        generic (
                T_pd : Time := unit_delay;
                T_ac : Time := unit_delay;
        );
        port (
                phi1 : in bit ;
                phi2 : in bit ;
                d_bus : inout bus_bit_8 BUS ;
                a_bus : out bit_12 ;
                reset : in bit ;
                ready : in bit ;
                read : out bit ;
                write : out bit ;
                result: out bit
        );
end mux2;
```

The entity has a generic constant, T_pd, used to specify the propagation delay between input events and output signal changes. The constant, T_ac, specifies the access times of registers. The default value, unit delay, is specified in a DSP_types

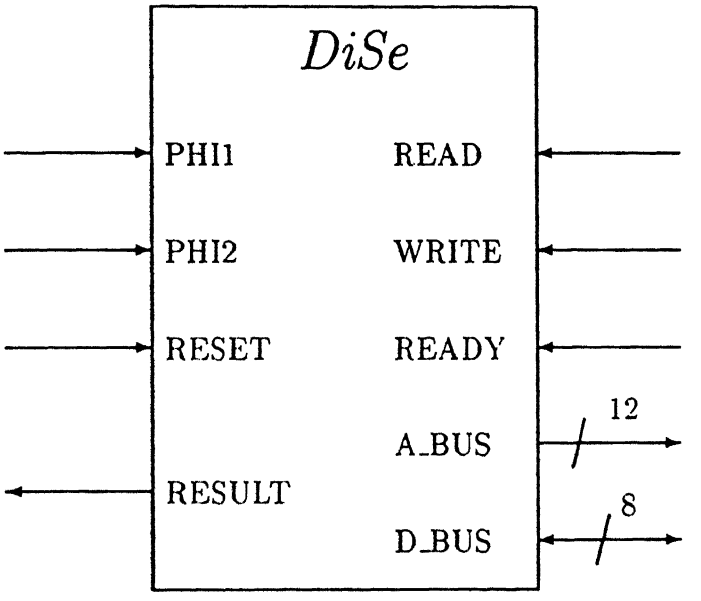


Figure 3.7: *DiSe* port interface

package. There are a number of ports corresponding to those shown in Figure 3.7. The reset, clocks (phi1 and phi2), bus control signals and result signal are represented by values of type bit. The address bus output is a simple bit-vector type as the processor is the only module driving the bus. On the other hand the data bus is a resolved bit-vector type, as it may be driven by both the processor and a memory module. The word bus in the port declaration indicates that all the drivers for the data bus may be disconnected at the same time (i.e, none of them is driving the bus).

Figure 3.8 shows the state diagram of the *DiSe* processor. Throughout its operation the *DiSe* traverses through a maximum of 10 states. Each one of these states can be broken down into smaller states as one goes into finer/lower levels of modeling. In the following sections we explain the behavior of the *DiSe* processor in each one of the ten states.

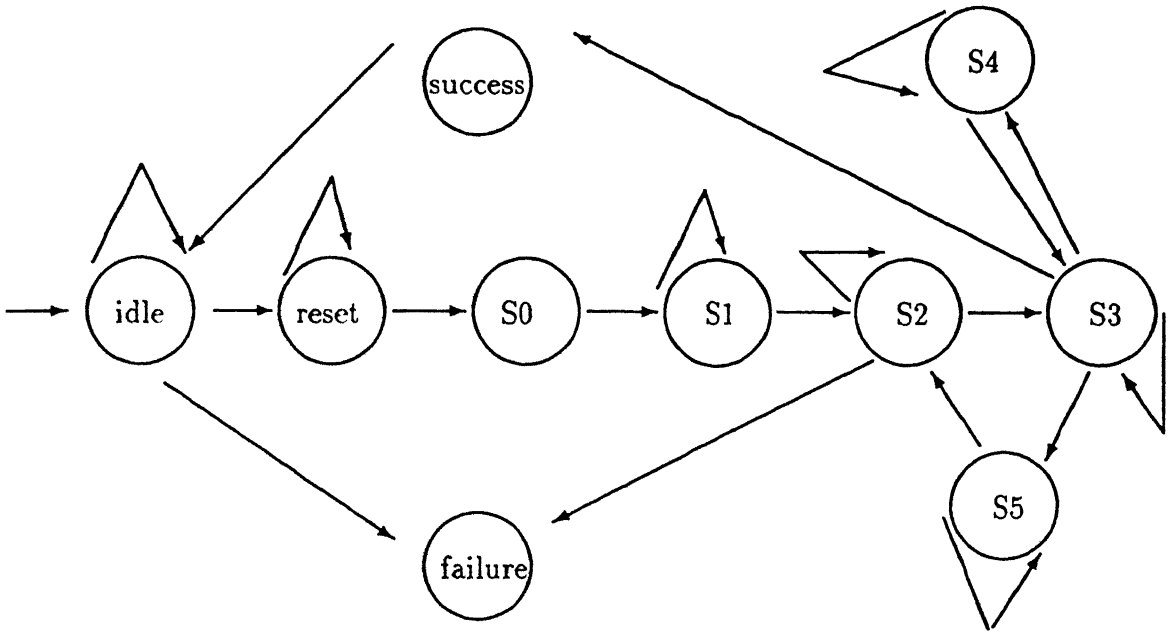


Figure 3.8: *DiSe* state transition diagram

The VHDL model is implemented by a single concurrent statement, an anonymous **process** which implements the behavior as a sequential algorithm. This process declares a number of variables which define the internal state of the processor.

3.8.1 Declarations for the *DiSe*

The declaration section for the architecture body contains declarations for

- The program counter.
- The general register file.
- The source register file.
- State variables to hold current and next state.

- Mux converters.
- Condition code flags.
- Other miscellaneous variables.

The procedure **memory_read**, implements the behavioral model of a memory read transaction. The parameters are the memory address to read from and a result parameter returning the data read. The procedure refers to the entity ports, which are visible because they are declared in the parent of the procedure.

The procedure **memory_write**, is similar, implementing the model for a memory write transaction. The parameters are the memory address to write to and the data to write. At the end of a transaction there is a null signal assignment to the data bus port. This models the behavior of the processor disconnecting from the data bus, i.e, it stops driving the data bus.

There are some other procedures which are called by the process implementing the *DiSe*. They are declared in the DSP_types package. The procedure **add** models the behavior of an adder. It takes two parameters, op_1 and op_2, which hold the two operands, a result parameter which holds the sum and an overflow parameter.

The procedure **compare** models the behavior of a comparator. It takes two parameters, op_1 and op_2, which hold the two values to be compared, and three flags, LT , EQ, GT which are set depending on whether op_1 is less than, equal to or greater than op_2.

The procedure **next_read** models the behavior of accessing the source register file. It takes as parameters the source register file, the next register which is to be read, the temporary variable into which the data will be placed and a flag which is set when the next register to be read points beyond the last register.

The procedure **next_write** models the behavior of writing into the source register file. It is similar to the next_read procedure. In both these procedures the controller

doesn't specify the address of the next procedure to be read or written. This information is maintained in the register file itself. This is possible since the source register file is always accessed sequentially, i.e the first register followed by the second register and so on. The count wraps around after reading the sixteenth register.

3.9 Controller Process

The body of the process which implements the controller is discussed next. This process is activated during the initialization phase of a simulation.

When powered up, the *DiSe* processor checks for the reset signal active at the rising edge of *phi1*. When the reset input is asserted, all of the control ports are returned to their initial states, the data bus driver is disconnected, the program counter is cleared, the next register to be fetched from the source register file is set to '0', result is set to '0' and the current state is set to **RESETTING**.

3.9.1 State : RESETTING

In this state the model waits until reset is negated before proceeding further. The model checks for the reset signal going '0' at every falling edge of phi2. If reset goes to '0' then the current state changes to *S_0*. Throughout the duration of operation of the *DiSe* processor, the reset input is checked after each bus transaction. If the transaction was aborted by a reset being asserted, no further action is taken in fetching or manipulating data. The control falls through to the reset handling code.

3.9.2 State : S_0(Fetch Input Word Length)

In this state the program counter, PC, contents are latched onto the address bus. The PC contents at this stage are X"000". As shown in Figure 3.3 memory location X"000" contains the length of the source(input) word, i.e the word which is being

searched. The source word length is fetched and stored in the SRC_LENGTH register in the general register file. The source word length is 8-bits long and so is the register SRC_LENGTH. If in the meanwhile reset is asserted then all operations are suspended and state changes to RESETTING. In the course of normal operation next state is changed to *S_1*.

3.9.3 State : S_1(Fetch the Input Word)

In this state the *DiSe* processor fetches a maximum of sixteen source word bytes from memory. Memory locations X"000" through X"100" are reserved for the source word. Each source word byte fetched from memory is stored in a register in the source register file. The source register file has sixteen registers. These are used to cache the source word bytes. The logic behind this is that when the source word bytes are compared with the words in the dictionary they need not be fetched from main memory every time they are compared. The *DiSe* uses the source register file to establish a sort of working set of source word bytes. It is expected that the first sixteen bytes of two words are enough to distinguish between them.

The SRC_LENGTH register is used extensively in this phase. The *DiSe* processor compares the SRC_LENGTH register contents against the PC to determine the number of bytes already fetched. The comparison also ensures that the number of bytes fetched is just enough to fit into the source register file.

The model changes state to *S_2* when it finishes fetching all the source bytes, in case the SRC_LENGTH is less than or equal to sixteen. If the SRC_LENGTH is exactly sixteen bytes then a flag *cc_L* is set. The utility of this flag will become clear shortly. In case the SRC_LENGTH happens to be greater than sixteen the *DiSe* processor just fetches the first sixteen bytes to source register file and changes state to *S_2*. The PC at this stage contains the number of source word bytes fetched. This is an offset into the source word string. The PC value is stored into a register

SRC_OFFSET. The value X"101" is loaded into the PC before switching to state *S_2*.

At the end of state S_1 we have

- SRC_LENGTH contains the source word length.
- SRC_OFFSET contains the number of source bytes fetched.
- The source register file contains source word bytes.
- PC is loaded with value X"101" and
- Next state is *S_2*.

3.9.4 State : S_2(Compare Word Lengths)

The *DiSe* processor fetches the length of the current word being compared from the dictionary. This byte is stored in the register CUR_LENGTH. It compares the current word length, which happens to be one byte long, with the length of the source word which is stored in register SRC_LENGTH. If the two lengths are not equal then the words cannot be equal and therefore PC is adjusted to point to the beginning of the next word. If the two lengths are equal then it is possible that they may be identical and hence it is necessary to compare them. The CUR_LENGTH is added to the PC contents and the sum is stored in a temporary register TEMP1. The next state is set as *S_3*.

It is also possible that the word being searched doesn't exist in memory. There must be some way for the *DiSe* to know that it has looked up all words in memory and it must not look further. This is accomplished by setting the word length in the last record to '0'. In state S_2, when the *DiSe* finds a record with word length '0' it moves to state *FAILURE*.

At the end of state S_2

- Next state is FAILURE or
- NEXT state is S_3 and
- PC points to the beginning of a dictionary record
- TEMP1 contains PC + CUR_LENGTH

3.9.5 State : S_3(Compare Words Bytewise)

The *DiSe* fetches bytes from the record currently pointed to by PC. The PC is incremented after each memory access. The contents of TEMP1 are compared with PC after each memory access to see if there are any more bytes from the current record that need to be fetched. Each byte fetched from memory is compared with a byte from the source register file. A match entails another memory fetch. If all the current record bytes match all the source word bytes then the *DiSe* enters the state *SUCCESS*.

If the source word and the current word are not equal then the first byte in which they do not match will cause the *DiSe* to return to the state *S_2*. However, before returning to S_2, the PC is adjusted to point to the next record in memory.

In some rare cases it may happen that the source word and the current word match for the first sixteen bytes. Now the cache must refreshed with subsequent bytes (seventeenth byte onwards) from the source word. The PC contents are stored in register CUR_LENGTH. The contents of the register SRC_OFFSET are loaded onto PC. Now the PC points to the next source word byte which should be fetched. The state is then changed to *S_4*.

Sometimes it may happen that a mismatch occurs between a source byte and a current word byte after the source register file has been refreshed at least once. Then the *DiSe* processor changes state to *S_5*.

3.9.6 State : S_4(Refresh Cache With Next Page)

In this state the *DiSe* processor goes into loop to refresh the source register file. The PC points to the next byte to be fetched. Each source byte fetched is stored in the source register file. After each memory access the PC is compared with the SRC_LENGTH register to determine if another byte is to be fetched. This loop continues until contents of PC equals contents of SRC_LENGTH or the source register file is full. Since this state was entered as a result of the successful match of all the bytes in the source register file the *DiSe* must return to state S_3 to continue its operation of matching corresponding bytes. Before returning to S_3 the PC contents are dumped into the SRC_OFFSET register. The PC is loaded with the contents of CUR_LENGTH so that it once again points to the byte in the current word which must be fetched next. This is done to restore the state in S_3 at the point at which the processor entered S_4.

The flag *cc_L* is set if the sixteenth byte in the source register file happens to be the last byte of the source word. The controller process checks this flag to determine if it must refresh the source register file.

3.9.7 State : S_5(Fill Cache With First Page)

This state is entered when the source word and the current word differ after the source register file has been refreshed at least once. The current record has to changed as a result of the mismatch. Now it is not possible to match the first byte of the current word with that of the source word. This is because the source register file contains bytes numbered higher than sixteen as a result of the refresh(es). So the source register file must be reloaded with the first sixteen bytes of the source word.

The operations in this state are identical to that of state S_1. The source register file is refreshed and then control is transferred to state *S_2*.

3.9.8 State : SUCCESS

This state is entered when the *DiSe* is successful in finding a matching word in the dictionary. The register TEMP1 points to the base address of the current record at all times. The lower byte of TEMP1 is written to the memory location X"000". The upper byte of TEMP1 is written to memory address X"001". The result signal is asserted to interrupt the master processor. The master processor then picks up the address of the matched word from the memory addresses X"000" and X"001", and goes to the *idle* state.

3.9.9 State : FAILURE

This state is reached when none of the words in the dictionary match the source word. The memory locations X"000" and X"001" are filled with zeroes and the result signal is asserted. The zero code is an indication to the master processor that the search was unsuccessful. The next state is the *idle* state.

3.9.10 State : IDLE

This is an idle state for the machine. It goes into a loop checking for the reset signal at every leading edge of*phi1*. If reset is asserted it goes into state RESETTING.

3.10 Register Transfer Architecture

This section deals with the internal structure of the *DiSe* processor. Such a description is invaluable, as it provides the ASIC designer with a set of handles or tunable parameters. The designer can then play around with these parameters and analyze the performance to arrive at an optimal design. This is very convenient to have before committing expensive resources to detailed design and implementation.

Another cogent reason is that the state of the art tools are not mature enough to synthesize gate level net-lists directly from the behavioral description.

Once a behavioral architecture has been decided upon, we carry out the design of the next level of architecture. Figure 3.9 shows a block diagram of a simple architecture to implement the *DiSc* processor. Such a hierarchical design simplifies the design process. The visible components are

- A twelve bit carry lookahead adder
- An eight bit comparator
- A general register file
- A source register file
- A program counter
- A control unit and
- Latches and muxes

The following subsections describe each of the components used in the proposed *DiSc* architecture. A VHDL entity description for each component is given in appendix A.

3.10.1 Multiplexor(M1 and M2)

The entity has a *select* input bit, an 8-bit wide input *i0*, a 12-bit wide input *i1* and a bit-vector output *y*.

The architecture body contains a VHDL concurrent **select** statement. It uses the value of the *select* input to determine which of the two bit-vector inputs is passed through to the output. The assignment is sensitive to all the input signals, so when any of them changes, the assignment will be resumed.

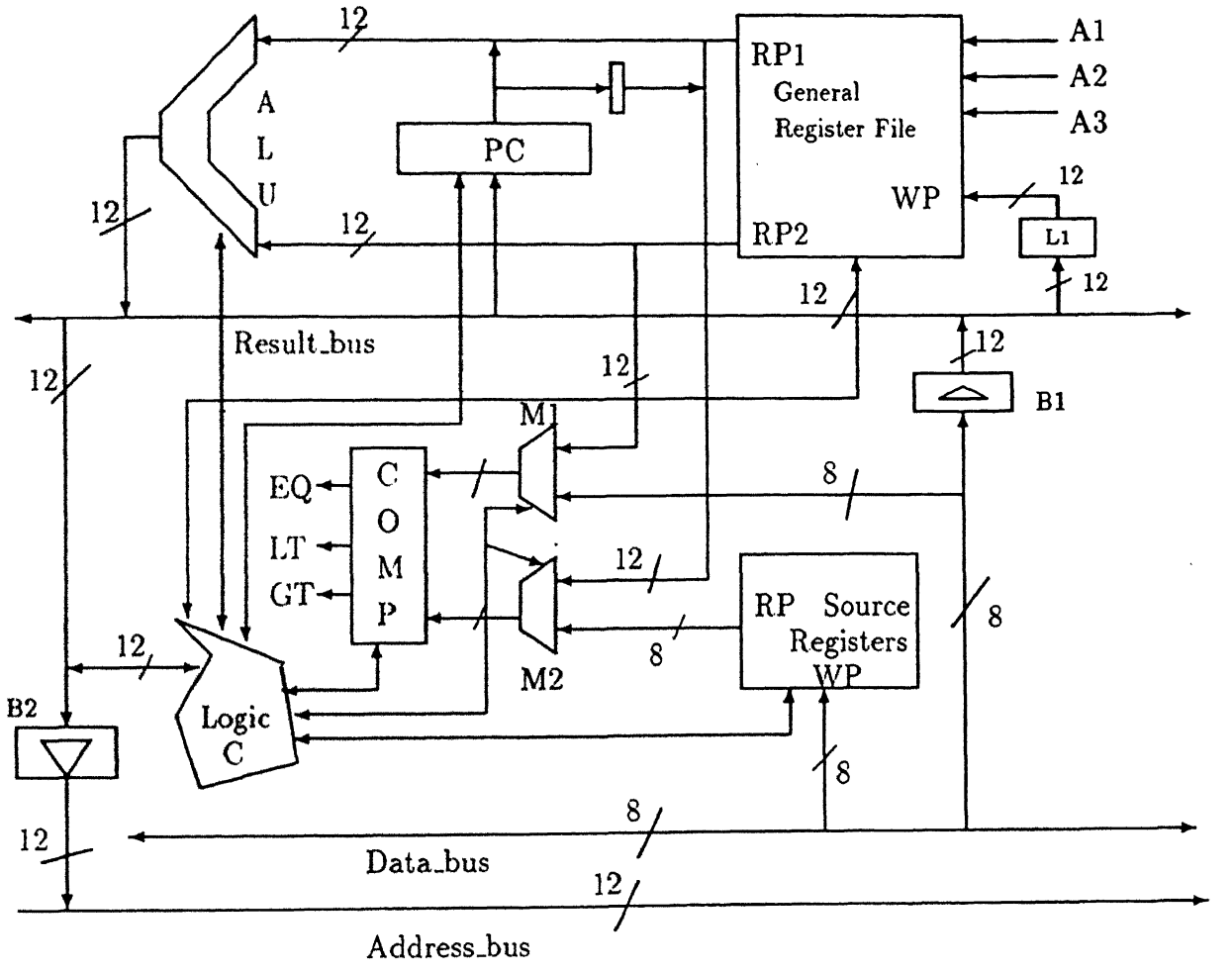


Figure 3.9: RTL architecture of the *DiSc* processor

3.10.2 Transparent Latch(L1, EQ, LT, GT)

The entity has an enable input bit *en*, a bit-vector input *d* and a bit-vector output *q*. The size of the bit vector ports is determined by the generic constant *width* which must be specified when the entity is used in a structural description. The entity models the behavior of a D-latch.

The architecture body contains a VHDL **process** statement sensitive to the *d* and *en* inputs. The behavior of the latch is such that when *en* is '1', changes on *d* are transmitted through to *q*. When *en* changes to '0', any new value on *d* is ignored an the current value on *q* is maintained.

3.10.3 Buffer(B1 and B2)

The entity has an enable input bit *en*, a bit-vector input *a* and a resolved bit-vector output *b*.

The behavior is implemented by a VHDL **process** statement sensitive to the *en* and *a* inputs. If *en* is '1' then the *a* input is transmitted through to the *b* output. If *en* is '0', the driver for *b* is disconnected and the value on *a* is ignored.

3.10.4 Program Counter(PC)

It has a 12-bit-vector input port *d_in*, an output port *d_out*, a *clear* bit input port, a *latch_en* and an *out_en* port.

Internally, the PC is implemented as a master slave register pair. When *clear* is enabled the PC is filled with zeroes. When *latch_en* is '1' then the value on *d_in* is stored in the master register but the output (if enabled) is driven from the previously stored value of the slave register. With *latch_en* disabled the slave value is updated from the master value and any further changes in *d* are ignored.

3.10.5 General Register File

It has two read ports *rp_1* and *rp_2* with their corresponding address ports *rad_1* and *rad_2*. It has only one write port *wp_gr* with its corresponding address port *a3*. Each one of the read and write ports has an enable port associated with it.

The behavior is implemented by a single VHDL process statement. When any of the inputs change, firstly the write port enable pin, *en3*, is checked, and if asserted, the addressed register is updated. Then the read ports are checked. If asserted the addressed data is fetched and driven onto the corresponding data bus. If the port is disabled then the data output bus driver is disconnected.

3.10.6 Source Register File

It has just one read port *RD* with an enable port *en1*, a write port *WR* with enable port *en2* and three bit ports, *next*, *init* and *count_16*.

The behavior is implemented as a single VHDL process. The *init* signal when asserted, resets the internal state of the source register file. The next register to be accessed is set to the first register. For every *next* signal the internal register pointer is incremented. The signal *count_16* is asserted when the internal register pointer "falls off" after pointing to the last register. It actually wraps around to point to the first register.

3.10.7 Arithmetic Unit(ALU)

The two 12-bit-vector ports *op_1* and *op_2* hold the two operands. *Result* port holds the result of the arithmetic operation. The *command* port tells which function to perform while the *enable* port fires the operation. Port *carry_in_0* holds the carry into the zeroth bit for an add operation.

The Arithmetic unit supports only two functions :

1. Adds op_1 and op_2 when command is '1'

2. Simply passes op_1 through to result when command is '0'

The function is performed only when *enable* is asserted. The result bus is only driven if enable is asserted, otherwise it is disconnected.

The adder is implemented as a carry-lookahead adder. Hamacher et. al., 1990, contains a detailed description of the carry-lookahead adder. This implementation was chosen mainly because it its speed doesn't suffer with scaling. This adder has a theoretical complexity of only 6 gate delays for producing the sum and carry-out from the most significant bit. However practical considerations like fan-in and fan-out prevent the processor from delivering the expected performance of 6 gate delays. So the implementation assumes a fan-in of 4. The 12-bit adder is then implemented using three 4-bit carry-lookahead adders. Each one of the 4-bit adders takes 3 gate delays to produce all the carries. A further 2 gate delays are required to generate C_12 by applying the carry lookahead technique over the carries produced by the 4-bit units. After all the carries have been produced, 3 more gate delays are required to produce the sum. Thus the output is available 8 gate units after the inputs have been applied.

3.10.8 Comparator Unit(COMP)

The two 8-bit wide bit-vector ports *op_1* and *op_2* hold the two values to be compared. The bit ports *EQ*(equal), *LT*(less) and *GT*(greater) show the results of the comparison, and *enable* is used to fire the comparison.

The architecture consists of a single VHDL process statement. Once *enable* is asserted, *op_1* is compared with *op_2*. *EQ* is asserted if *op_1* is equal to *op_2*, *GT* is asserted if *op_1* is greater then *op_2*, and *LT* is asserted if *op_1* is less than *op_2*.

A 12-bit comparator, assuming 4-input gates, requires 4 gate delays to produce the *EQ* output. The number of gates required is - 12 X_NOR gates and 3 AND gates. The delay required to produce the *LT* output is 6 gate delays. The number of gates required for the *LT* output is - 12 NOT gates, 12 XNOR gates, 3 OR gates, and 36 AND gates. The same is true also for the *GT* output.

All these components can be connected in a structural VHDL model to yield a structural description.

3.11 Future Directions

The design of the *DiSe* processor presented above was modeled at the behavioral level. There are many intermediate levels of modeling before it can be delivered to the fabrication unit. However, detailed design can build on the framework provided by our behavioral model. Here we suggest a couple of improvements that may appreciably improve the performance of the *DiSe*.

3.11.1 An Effective Pre-comparison Criteria

As shown in Figure 3.3, each word in the dictionary is prefixed with its length. Besides acting as a record delimiter it also serves another purpose. The *DiSe* compares this length with the source word length. If the two lengths are equal then words need to be compared. A better method would be to reserve a byte next to the length field. An effective hash function can be used to calculate a *unique* seed value for each word which is then stored in the reserved byte. This would significantly speed up the comparison process as the *DiSe* can then home onto the target word faster by avoiding the comparison of different words of same length. Note that the seed value need not be unique for all the words in the dictionary. It

must be *unique* for all the words in a single block or partition of the dictionary. The modified record structure is shown in Figure 3.10.

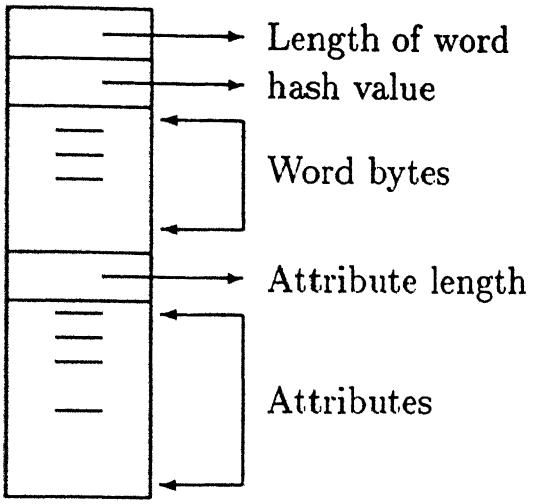


Figure 3.10: Modified dictionary record structure

3.11.2 Improving The Order of the Search Algorithm

The search algorithm employed by the *DiSe* has a complexity of O(n), where n is the number of words in the dictionary block loaded into memory. By changing the architecture slightly we can reduce the search algorithm complexity to $O(\lg n)$ (employ binary search). For binary search the data must be sorted. Instead of placing the words in a sorted order we introduce a new data-structure which gives a sorted view of the words. The memory organization, shown in Figure 3.11, changes slightly with the introduction of this data structure. It is basically an array of pointers to records. This data structure is preceded by its length. A simple binary search can now be implemented.

During the course of the binary search the upper and lower bounds of the array keep changing. At each stage calculation of a new bound requires a division by 2

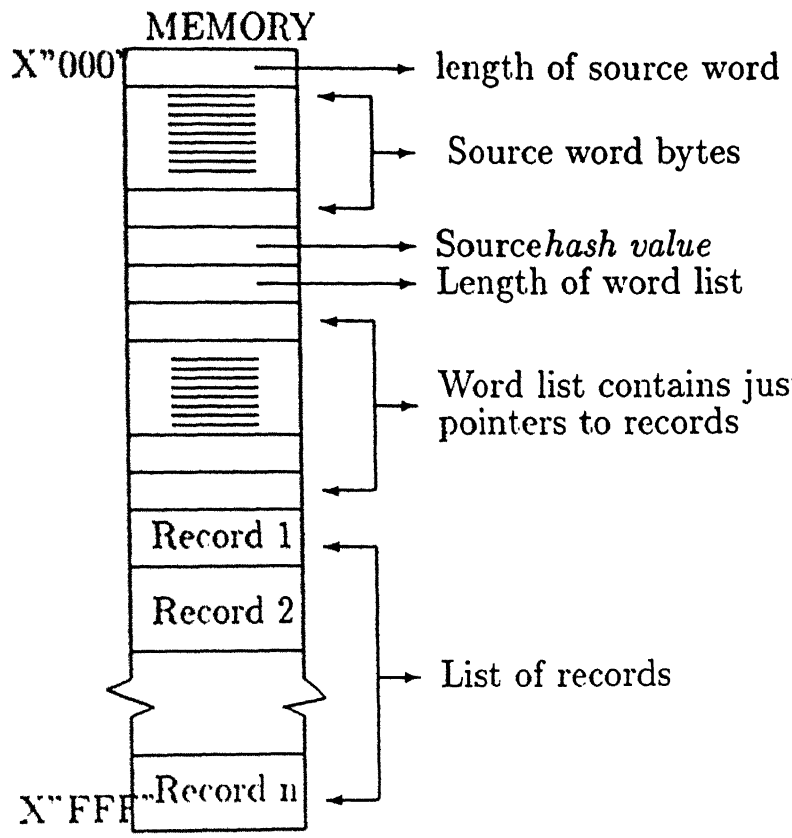


Figure 3.11: Input data format for binary search

operation. This can be implemented by a simple right shift of the dividend. There is no need to introduce a new divider unit.

3.11.3 Implementing n−way Comparison

In the present implementation, the input and the dictionary words are compared byte by byte. This is because the data bus is only 8−bits wide and can only fetch a byte at a time. By using a 64-bit data bus 8 bytes can be fetched at a time. The comparator must now be capable of comparing two 64-bit operands at a time. There is a high probability that if two words are different then the mismatch may be detected in the first 64 bits(at least for English language words). In most cases

we may be able to decide on the equality of the words in just one comparison.

On the average, half of the memory (n/2 words) will be checked for every search. Assuming a normal data distribution, this implementation requires only n/2 comparisons. The number of comparisons can be improved to a worst case Figure of $\lg n$ if this method is used in conjunction with the binary search method mentioned earlier.

Of course, this concept can be extended to data-bus bandwidths of up to 128 bits as the internal source-word-cache can cache up to 16 input-word bytes.

3.12 Pseudo-code for the *DiSe* Model

This section presents, in pseudo-code, the various operations performed by the DiSe processor, in each state.

3.12.1 State : RESETTING

1. Wait for falling edge of phi2
2. If reset is '0' then goto state S_0
3. goto 1

3.12.2 State : S_0

1. Fetch byte from memory_location X"000"
2. Store in SRC_LENGTH register
3. goto S_1

3.12.3 State : S_1

1. Increment PC
2. Read byte from memory location pointed to by PC

3. Write the byte into source register file
4. If *count_16* is asserted then
   - Save PC in SRC_OFFSET register
   - If SRC_LENGTH is equal to 16 set cc_L flag
   - Load PC with X"101" and reset count_16
   - goto S_2
5. If *count_16* is not asserted then
   - If PC value is less than SRC_LENGTH goto 1 else
   - Save PC in SRC_OFFSET register
   - Load PC with X"101" and reset count_16
   - goto S_2

3.12.4 State : S_2

1. Fetch the *length* of the current word
2. If *length* is not equal to SRC_LENGTH goto FAILURE
3. If *length* is equal to SRC_LENGTH then
   - store *length* in CUR_LENGTH register
   - if CUR_LENGTH is not equal to SRC_LENGTH then

     Adjust PC to point to attribute length

     Fetch attribute length

     Adjust PC to point to beginning of next record

     goto S_2
   - if CUR_LENGTH is equal to SRC_LENGTH then

     Store PC in TEMP1 register

     Store PC + CUR_LENGTH in TEMP2

     goto S_3

3.12.5 State : S_3

1. Increment PC by 1
2. Fetch a cur_byte from the current word
3. Fetch a Src_byte from source register file
4. If src_byte is not equal to cur_byte then
   - If source register file has to be refreshed then

     Store PC contents in CUR_LENGTH

     goto S_5
   - If source register file doesn't need a refresh then

     Adjust PC to point to next record

     goto S_2
5. If src_byte is equal to cur_byte then
   - If this was the last byte then goto SUCCESS
   - If this was last byte in source register file then

     Store PC contents in CUR_LENGTH

     goto S_4
   - If this was not the last byte then goto 1

3.12.6 State : S_4

1. Increment PC by 1
2. Fetch a source word byte from memory
3. Store the byte in source register file
4. If source register file is full then
   - Store PC in SRC_OFFSET register
   - Restore status of PC to pre S_4 entry value
   - goto S_3

5. If source register file is not full then

   - If all source word bytes have been fetched then

     Store PC in SRC_OFFSET register

     Restore status of PC to pre S_4 entry value

     goto S_3

   - If all source word bytes have not been fetched then goto 1

3.12.7 State : S_5

1. Increment PC by 1
2. Read a source from memory
3. Store the source byte in source register file
4. If source register file is full then
   - Store PC in SRC_OFFSET register
   - Restore status of PC to pre S_5 entry value
   - Adjust PC to point to next record
   - goto S_2
5. If source register file is not full then
   - If all source word bytes have been fetched then

     Store PC in SRC_OFFSET register

     Restore status of PC to pre S_5 entry value

     Adjust PC to point to next record

     goto S_2

   - If all source bytes have not been fetched goto 1

3.12.8 State : SUCCESS

1. Write the lower 8 bits of TEMP1 into memory X"000"

2. Write the upper 8 bits of TEMP1 into memory X"001"
3. Set result port to '1'
4. goto state IDLE

3.12.9 State : FAILURE

1. Write the value X"0" into memory X"000"
2. Write the value X"0" into memory X"001"
3. Set result port to '1'
4. goto state IDLE

3.12.10 State : IDLE

1. Do nothing until reset is asserted

2. Write the upper 8 bits of TEMP1 into memory X"001"
3. Set result port to '1'
4. goto state IDLE

3.12.9 State : FAILURE

1. Write the value X"0" into memory X"000"
2. Write the value X"0" into memory X"001"
3. Set result port to '1'
4. goto state IDLE

3.12.10 State : IDLE

1. Do nothing until reset is asserted

Chapter 4

Test Bench For The *DiSe* Processor

4.1 Introduction

This chapter describes the design of a test bench circuit that was used in the testing of the *DiSe* processor. Figure 4.1 shows the organization of the test bench circuit. The test bench was modeled in VHDL.

The test bench consists of three main components

1. A clock_generator device
2. A memory device and
3. The *DiSe* processor

The clock generator device generates the two-phase clock and the reset signal to drive the processor. The memory device stores the test data. The behavioral models of all three components were connected together in a structural description of the test bench.

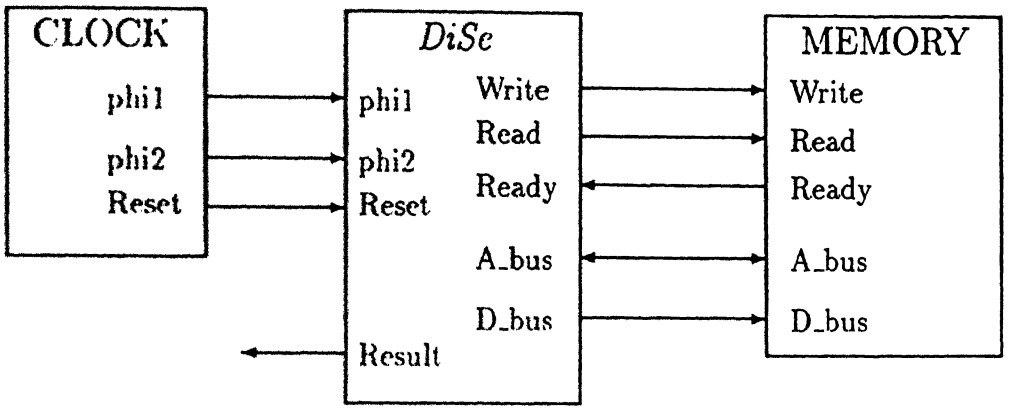


Figure 4.1: Test bench circuit for the *DiSe*

4.2 Clock Generator

The clock generator has two formal generic constants. T_pw is the pulse width for each of phi1 and phi2, i.e, the time for which each clock signal is '1'. T_ps is the pulse separation, the time between one clock signal changing to '0' and the other clock signal changing to '1'. Based on these values, the clock period is twice the sum of the pulse width and the separation.

The architecture of the clock generator consists of two concurrent statements, one to drive the *reset* signal and the other to drive the clock signals. The reset driver schedules a '1' value on reset when it is activated at simulation initialization, followed by a '0' a little after two clock periods later. This concurrent statement is never subsequently reactivated, since its waveform list does not refer to any signals. The clock driver process, when activated, schedules a pulse on phi1 immediately, followed by a pulse on phi2, and then suspends for a clock period. On resuming, it repeats scheduling the next clock cycle.

4.3 The Memory Device

The architecture body consists of one VHDL process statement to implement the behavior. The process contains an array variable to represent the storage of the memory. When the process is activated, it places the output ports in an initial state where the data bus is disconnected and the ready bit is negated. It then waits for a read or write command. When one of these occurs, the address is sampled and converted from a bit vector to a number. If it is within the address bounds of the memory, the command is acted upon.

For a write command, the ready bit is asserted after a delay representing the write access time of the memory, and then the model waits until the end of the write cycle. At that time, the value that existed on the data bus a propagation delay before the current time is sampled and written into the memory array. The use of this delayed value models the fact that memory devices actually store the data that was valid before the triggering edge of the command bit.

For a read command, the data from the memory array is accessed and placed on the data bus after a delay. This delay represents the read access time of the memory. The ready bit is also asserted after the delay, indicating that the processor may continue. The memory then waits until the end of the read cycle.

At the end of a memory cycle, the process repeats, setting the data bus and ready bit drivers to their initial states, and waiting for the next command.

4.4 Test Bench Circuit

Figure 4.1 shows the structural architecture of the test bench circuit. The entity contains no ports, since there are no external connections to the test bench. The architecture body contains component declarations for the clock driver, the memory and the *DiSe* processor. The ports in these component declarations correspond

Table 4.1: Test Data No. 1

Memory Location		Word	Length
From	To		
0001	0005	Heart	5
0007	0011	Heard	5
0016	0019	Head	4
0023	0027	Heart	5

Heart. The state of the processor and the bus transactions at various points of time are shown in table 4.2. The first column shows the time at which the processor entered a particular state(shown in the last column). The second column shows the value on the address bus at that instant and the third column shows the value on the data bus at the same instant. The fourth column shows the time till which the processor remains in the particular state.

The length(entry in column 6) of the input word is fetched first as shown in the third row of table 4.2 followed by the search word itself as shown in the fourth row. The first word in the dictionary area is "Heard". Since its length matches the search word length, the processor proceeds to match the two words(row 6). A mismatch occurs in the last letter. The second word, "Head", is only 4 bytes long and thus is skipped(row 7). The third word, "Heart", is of the same length as the search word so the processor proceeds to compare it with the search word(row 9). The two words match byte for byte. The search terminates in the SUCCESS state(row 10). The result pin is set to 1, and the processor goes into an idle state waiting for the next word to be searched.

This test case was designed to check the very basic functionality of the *DiSe* processor. The results indicate that the model works for all words which are less than or equal to sixteen bytes in size.

Table 4.2: Simulation Trace of Data No. 1

Time	A_Bus	D_Bus	Time	A_Bus	D_Bus	State
0	null	null	0	null	null	IDLE
0	null	null	47	null	null	RESETTING
48	0	null	120	0	5	S_0
162	1	H	521	5	t	S_1
541	6	0	601	6	5	S_2
642	7	H	1001	11	d	S_3
1081	15	0	1140	15	4	S_2
1221	22	0	1281	22	5	S_2
1322	23	H	1681	27	t	S_3
1731	0	22	-	1	0	SUCCESS
-	0	0	-	0	0	IDLE

4.5.2 Test Case 2

Table 4.3: Test Data No. 2

Memory Location		Word	Length
From	To		
0001	0255	x(49)Hx(204)H	255
0257	0511	x(255)	255
0521	0775	x(49)Hx(204)H	255

The *DiSc* processor was run on the test data shown in table 4.3. The simulation trace is shown in table 4.4. The search string is 255 bytes long. A page of the source word data includes 16 source word bytes. The first 16 bytes(page) of the source word are fetched and stored in the source cache. The first word in the dictionary area is also 255 bytes long. The processor matches the two words. After the first sixteen bytes, the source cache is loaded with the next page of source word bytes. A third page is loaded after the two words match in the first 32 bytes. A mismatch occurs in the 50th byte. The search begins afresh with the next word, with the

Table 4.4: Simulation Trace of Data No. 2

Time	A_Bus	D_Bus	Time	A_Bus	D_Bus	State
0	null	null	0	null	null	IDLE
0	null	null	47	null	null	RESETTING
48	0	null	120	0	255	S_0
162	1	x	1420	16	x	S_1
1421	256	0	1481	256	255	S_2
1501	257	x	2780	272	x	S_3
2781	17	x	4060	32	x	S_4
4061	273	x	5340	288	x	S_3
5341	33	x	6620	48	x	S_4
6621	289	x	7900	304	x	S_3
7901	49	x	9180	64	x	S_4
9181	305	x	9340	306	x	S_3
9341	1	x	10600	16	x	S_5
10601	512	0	10661	512	7	S_5
10681	520	0	10741	520	255	S_2
10761	521	x	12040	536	x	S_3
12041	17	x	14600	32	x	S_4
-	-	-	-	-	-	S_3/4
47881	240	x	49061	255	H	S_4
49081	761	x	50261	775	H	S_3
50281	0	0	50342	0	8	SUCCESS
50343	1	0	50402	1	2	SUCCESS
-	-	-	-	-	-	IDLE

source cache being loaded with the first page. This word matches the source word in all the 255 bytes. The search is ended with success.

In normal operation it is highly unlikely that such a data set will be encountered. It is designed to check the caching and paging operations of the source buffer. Paging is required only if the word lengths exceed 16 characters. Most of the characters in the words were set to x where an x represents a blank character. A series of n blank characters occurring in a word is represented as x(n) in the table. The results of the test run are tabulated in table 4.4 .

Chapter 5

The MiDiCal Processor

5.1 Introduction

This chapter discusses the architecture and design of the MiDiCal(Minimum Distance Calculation) processor. In the HEBMT system, the source language input sentence, IS, is compared against all the examples belonging to its syntactic category, to find the example which is closest in meaning to it. The example-base is partitioned on the basis of the syntactic categories of the sentences, i.e, all sentences belonging to one syntactic group go into one partition. This partition may further be partitioned on the basis of other sub-attributes. A HEBMT system assigns certain attributes to the IS, based on information gleaned in the previous phases. Based on these attributes a distance function is used to calculate the distance between the input sentence and each sentence of the example-base. The sentence with the minimum distance is output as the translation in the target language. For a full fledged HEBMT system the example-base contains a large amount of data. The partitions, with the best partitioning schemes, may still contain a lot of data. So the distance calculation phase is a bottleneck in the system. This bottleneck can be greatly reduced if the distance matching algorithm is implemented in hardware. This

chapter presents the architecture and design of an ASIC which can be used in speeding up the distance calculation algorithm. The chapter is organized as follows:

1. The distance calculation function.
2. Instructionless processor.
3. Input data format.
4. The MiDiCal processor interface.
5. The MiDiCal bus protocol.
6. Architecture Modeling: Component description.

5.2 The Distance Calculation Function

The distance between the input sentence, IS, and the example-base sentence is calculated according to the mathematical function given below :

$$d(IS, ES) = \sum_{p=1}^{n} d_p(ISG, ESG) + d_v(ISV, ESV) \tag{5.1}$$

where n is the number of noun syntactic groups in the IS and ISG and ESG are "input sentence noun syntactic group" and "example sentence noun syntactic group" respectively, and ISV and ESV are Input and Example sentence verb groups.

$d_p(ISG, ESG)$– distance between the pth noun syntactic groups of input and example source language sentence

$d_v(ISV, ESV)$– distance between the verb group of input and example source language sentence

$$d_p(ISG, ESG) = (w1 \times ATD + w2 \times ASD)/(w1 + w2)$$

$$ATD = (q1 \times SD + q2 \times (GD + ND + PD))/(q1 + q2)$$

$$d_v(ISV, ESV) = VCD$$

where ATD, SD, GD, ND, PD, ASD and VCD are attribute difference, status difference, gender difference, person difference, additional semantic difference, and verb category difference between example sentence and input sentence. w1,w2,q1,q2 are weights assigned to distance parameters according to their effect on translation. Additional semantic difference is retrieved from the semantic difference table. Additional semantic difference table is generated based on a hierarchy of semantic tags in the semantic network. The original paper on HEBMT(Jain et. al.[1]) contains a more detailed description of the distance calculation algorithm.

5.3 Instructionless Processor

The pros and cons of an instructionless processor versus an instruction-set processor have been discussed in chapter 3. The MiDiCal processor has been implemented as an instructionless processor. The reason is that the distance calculation algorithm lends itself to a very simple and straightforward datapath implementation.

5.4 Input Data Format

This section presents the format in which data must be fed to the MiDiCal processor. Figure 5.1 shows the memory organization. Note that the memory contains only abstracted examples. The memory basically contains

1. The abstracted input sentence and

2. Some source language patterns from the example-base.

The format of the input sentence is shown in Figure 5.1(b). The first word(4 bytes) are reserved for the BITMAP. The first n bits in the BITMAP are set to indicate the number of noun phrases in a sentence. The next six words are reserved

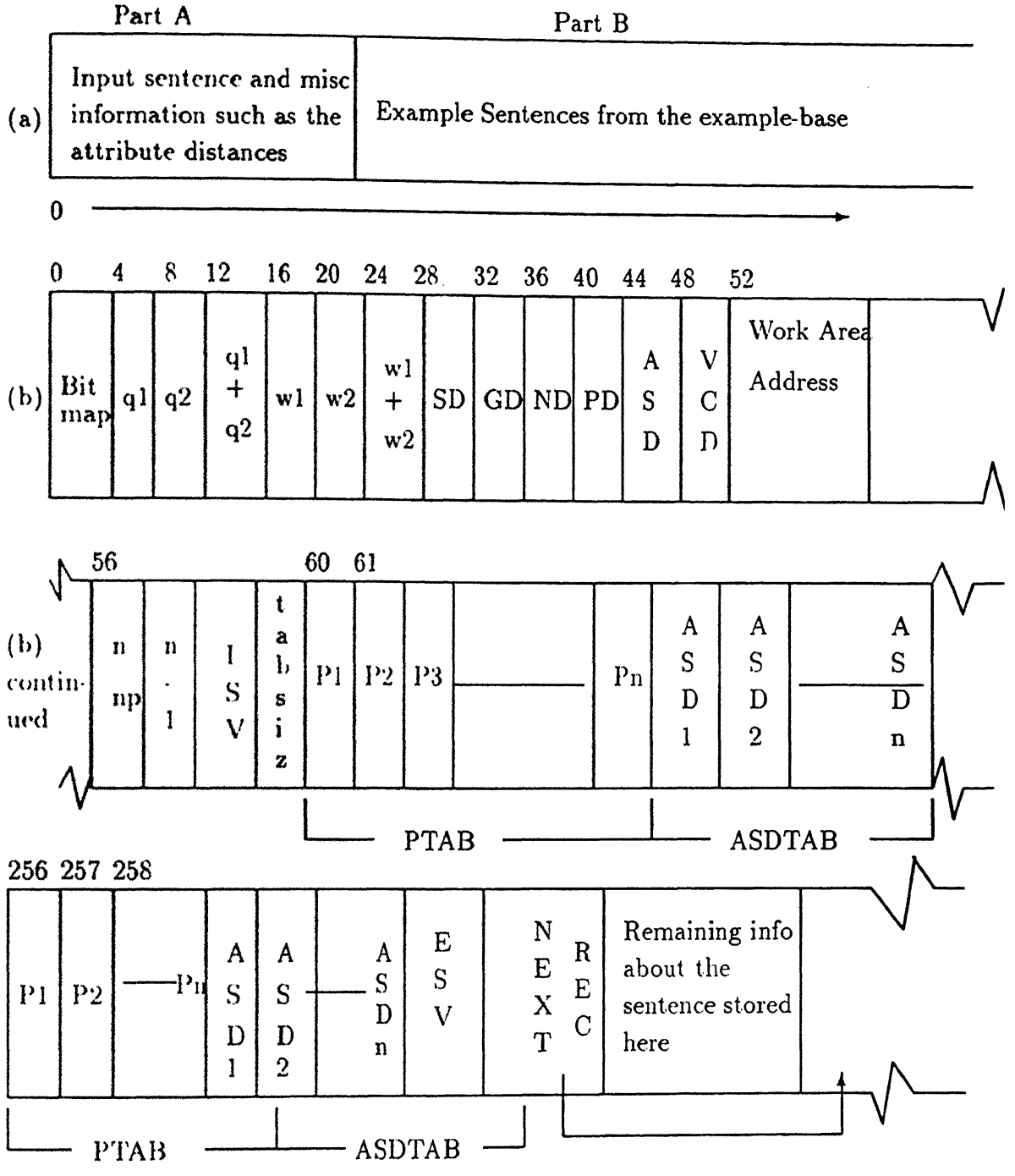


Figure 5.1: (a) Input data format (b) Part A format (c) Part B format

for the values of q1, q2, q1+q2, w1, w2, and w1−w2. The word (4 bytes) beginning at the 52nd byte holds the value of the WORK_AREA_ADDRESS. This is a pointer to a location in memory where the processor can write the results after finding the sentence with the minimum distance. The 56th byte holds n, the number of noun phrases. The 57th byte holds the value n−1. The next byte holds the input sentence verb property. The next byte contains the actual table size of noun phrases. This is followed by a table of noun phrase bytes and a table of additional semantic difference bytes(ASD).

The format of the example sentences from the example−base is shown in Figure 5.1(c). Each sentence begins with a table of noun phrase bytes followed by a table of corresponding additional semantic difference bytes. The tables are followed by a byte for the example sentence verb (ESV). Two more bytes hold a pointer to the next sentence in the example base. This field followed by some arbitrary number of bytes where any information about any of the noun phrases can be stored in any format as desired by the designers of the machine translation system.

The organization of the noun phrases needs some explaining. Each noun phrase is allocated one byte of memory. The attributes of the noun phrase are encoded in the bits of the byte as follows :

- Bit−7 and bit−6 represent the status(animate, inanimate, etc.) of the noun phrase.
- Bit−5 and bit−4 represent the gender(masculine, feminine, neutral) of the noun phrase.
- Bit−3 and bit−2 represent the number(singular, plural) of the noun phrase.
- Bit−1 and bit−0 represent the person(first person, second person, etc.) of the noun phrase.

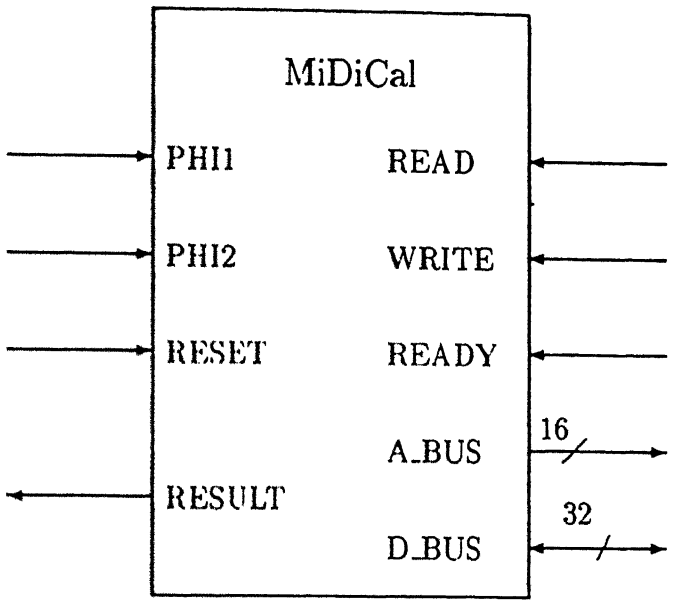


Figure 5.2: MiDiCal interface

5.5 The MiDiCal Processor Interface

The port diagram of a MiDiCal processor is shown in Figure 5.2. The processor interfaces to the memory module through a synchronous, unidirectional, 16 bit wide address bus. Data flows between the processor and the memory module through a bidirectional, 32 bit wide data bus. The read and write single bit ports are used to control read and write transactions. The ready port is used by the memory module to signal the availability of data on the data bus in a read transaction, and its readiness to accept data in a write transaction. The result port is used by the MiDiCal processor to signal a master processor that it has finished with the current set of data. The processor is driven by a two phase clock. The two ports, phi1 and phi2, provide the two phase clock.

5.6 MiDiCal Bus Protocol

The MiDiCal processor follows the same read and write bus protocols as the *DiSe* processor. The clock signal diagram, the bus read, and bus write timing diagrams are identical to the ones shown in chapter 3.

5.7 Architecture Modeling

This section discusses the architecture of the MiDiCal processor. The processor was modeled in the Verilog hardware description language. The processor was modeled structurally at a very high level and the components were modeled behaviorally. Figure 5.3 shows the architecture (block level) of the MiDiCal processor. There are 4 main blocks or units in the figure. Units B, C, and D are simple enough and their block diagrams show sufficient detail. Figure 5.4 shows a more detailed diagram of unit A.

The following are some of the main components used in the modeling of the processor

1. The logic controller
2. An integrated floating point addition and division unit
3. A floating point multiplication unit
4. A data fetch and dispatch unit
5. Register files.
6. a floating point comparator unit.
7. An integer addition unit.
8. A noun phrase comparator unit.

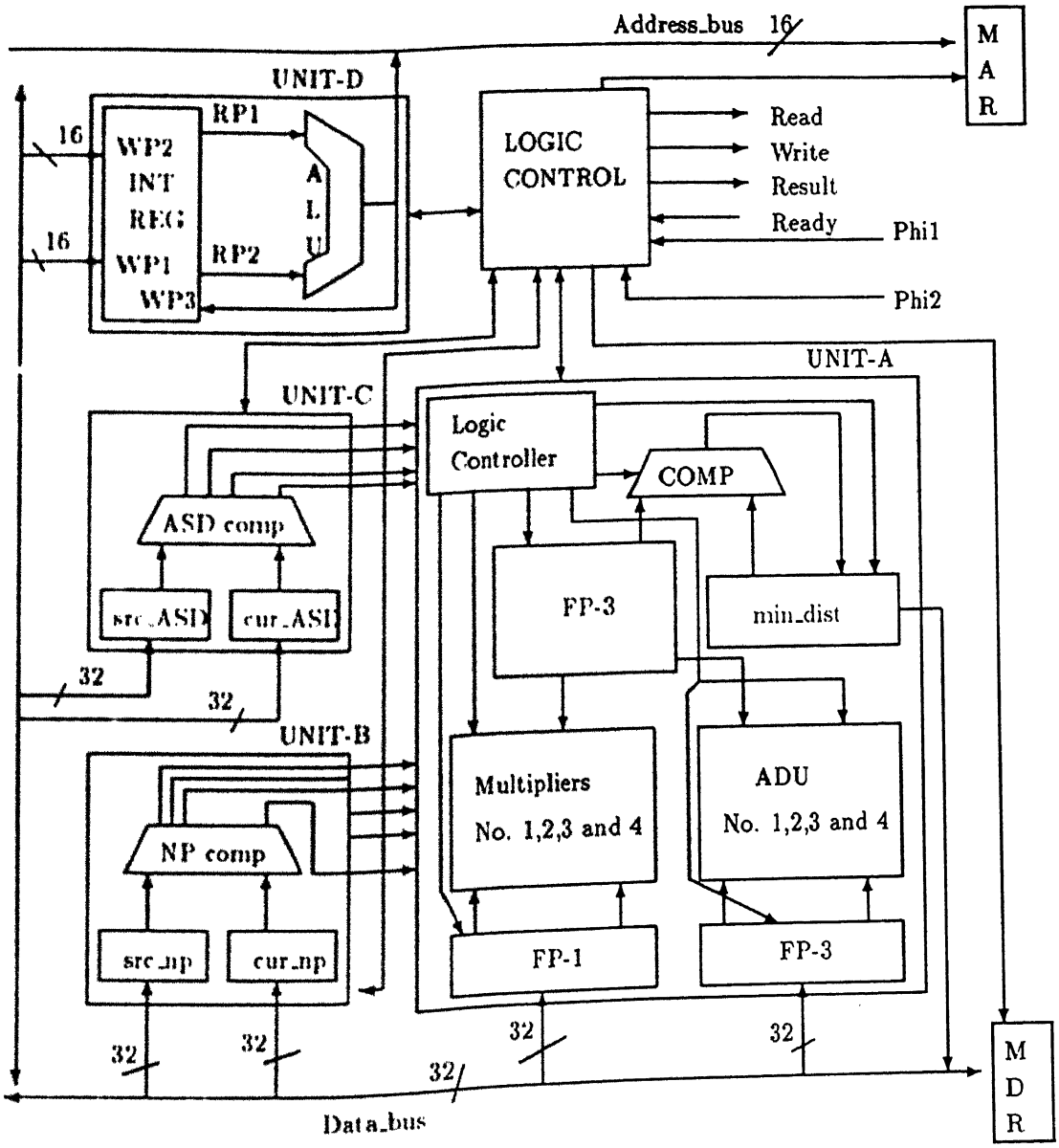


Figure 5.3: Architecture of the MiDiCal processor

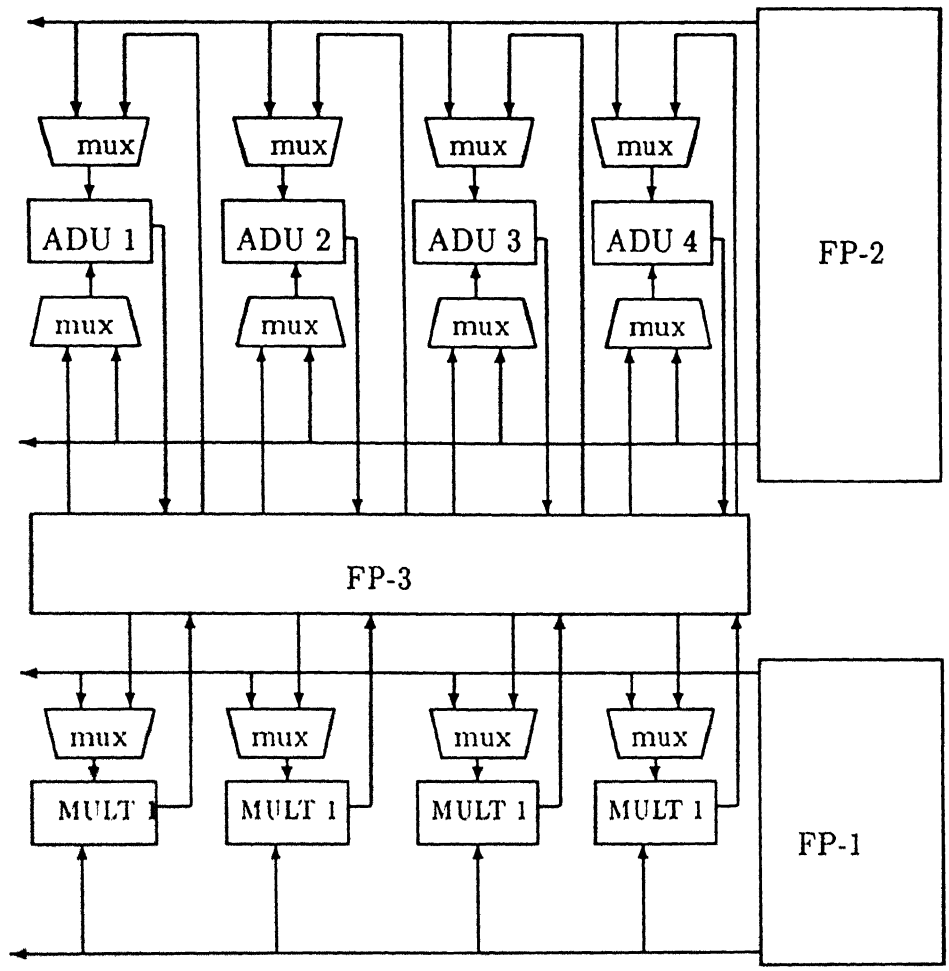


Figure 5.4: Detailed layout of UNIT-A

9. ASD comparator unit.

The logic controller has been implemented as a state machine. In the following discussion, the behavior of the MiDiCal processor is sketched algorithmically and the behavior of each component is described as and when needed.

5.7.1 The Logic Controller Unit

This unit provides control signals to all the components of the processor and coordinates all their functions. When the processor is powered on, it is placed in the *idle* state. The processor continuously loops in this state waiting for the reset signal to be asserted. Meanwhile the master processor loads the memory with some data and resets the MiDiCal processor. The logic controller puts the processor in the RESETTING state and waits for the negative edge of the reset signal. At this point, the processor moves to state S_0 and the actual working of the processor starts.

S_0 : In this state the processor gathers all the information about the current data set and stores that information in its internal registers. Some of this information is static in nature, i.e, doesn't change throughout the course of the processor's operation. The processor performs the following actions in this state :

1. Fetch the BITMAP string from memory[0:3] and store in register BITMAP.
2. Fetch the value of q1 from memory[4:7] and store in register REG_q1.
3. Fetch the value of q2 from memory[8:11] and store in register REG_q2.
4. Fetch the value of q1+q2 from memory[12:15] and store in register REG_qsum.
5. Fetch the value of w1 from memory[16:19] and store in register REG_w1.
6. Fetch the value of w2 from memory[20:23] and store in register REG_w2.
7. Fetch the value of w1+w2 from memory[24:27] and store in register REG_wsum.
8. Fetch the value of SD from memory[28:31] and store in register REG_sd.
9. Fetch the value of GD from memory[32:35] and store in register REG_gd.
10. Fetch the value of ND from memory[36:39] and store in register REG_nd.

11. Fetch the value of PD from memory[40:43] and store in register REG_pd.

12. Fetch the value of ASD from memory[44:47] and store in register REG_asd.

13. Fetch the value of VCD from memory[48:51] and store in register REG_vcd.

14. Fetch the value of Work_area_addr from memory [52:53].

15. Fetch the number of noun phrases from memory[53] and store in register NOF_np.

16. Fetch the value of ISV from memory[54] and store in register ISV.

17. Fetch the size of the table of noun phrases, in bytes, from memory[55] and store in register PTAB_SZ.

S_1: The process of distance calculation involves the comparison of certain attributes of the input sentence and the current sentence from the example–base. The noun phrase attributes of the input sentence are stored in a table called PTAB and the ASD for each noun phrase is stored in a table known as the ASDTAB. In this state, the processor sets up its internal registers to point to the PTABs and ASDTABs of the input sentence and the current sentence of the example–base. The steps followed are as under :

1. Set SRC_PTAB_BASE register to point to the base address of the input sentence noun phrase table.

2. Set SRC_ASDTAB_BASE register to point to the base address of the input sentence ASD table.

3. Set SRC_PTAB_OFFSET register to point to the first entry of the input sentence PTAB.

4. Set SRC_ASDTAB_OFFSET register to point to the first entry of the input sentence ASDTAB.

5. Set CUR_PTAB_OFFSET register to point to the base address of the PTAB of the first sentence in the example base.

6. Set CUR_ASDTAB_OFFSET register to point to the base address of the ASDTAB of the first sentence in the example base.

Now the processor is ready to fetch and compare the input sentence and a sentence from the example base.

S_2: The processor fetches 4 bytes from the input sentence PTAB and ASDTAB. The SRC_PTAB_OFFSET and the SRC_ASDTAB_OFFSET are incremented by 4. The same steps are repeated for the current sentence from the example base. Note that the processor fetches 4 bytes even if the number of noun phrases is less than 4. This is because the PTAB and the ASDTAB sizes are multiples of 4.

1. Fetch 4 bytes from input sentence PTAB. Store in register SRC_PROP_WORD.

2. Increment SRC_PTAB_OFFSET by 4.

3. Fetch 4 bytes from current sentence PTAB. Store in register CUR_PROP_WORD.

4. Increment CUR_PTAB_OFFSET by 4.

5. Fetch 4 bytes from input sentence ASDTAB. Store in register SRC_ASD_WORD.

6. Increment SRC_ASDTAB_OFFSET by 4.

7. Fetch 4 bytes from current sentence ASDTAB. Store in register SRC_ASD_WORD.

8. Increment CUR_ASDTAB_OFFSET by 4.

S_3 : The processor compares the four bytes in the register SRC_PROP_WORD with those in CUR_PROP_WORD. The above two registers contain noun phrases fetched from the input sentence and current sentence PTABs. It is not necessary that all the four bytes be noun phrases. Some of the bytes could be padded bytes to make the PTAB size an exact multiple of 4. For each noun phrase, the four properties viz. SD, GD, PD, ND, are checked for equivalence. The results are stored in flags SD(1:4), GD(1:4), PD(1:4), ND(1:4). A flag is set to 1 if the corresponding properties are equal.

Similarly, the ASD tags stored in CUR_ASD_WORD and SRC_ASD_WORD are compared. The flags F_ASD1_EQ, F_ASD2_EQ, F_ASD3_EQ and F_ASD4_EQ are set to 1 if the corresponding tags match.

S_4 : The MiDiCal processor has 4 multipliers and 4 ADUs (Addition and Division units) internally. As can be seen from equation 1, only three operations are involved in the distance calculation process- addition, division and multiplication. Since the processor processes a maximum of only 4 noun phrases at a time, each noun phrase is assigned one multiplier and one ADU. As mentioned earlier, the number of noun phrases may be less than 4. In such situations the processor disables those multipliers and ADUs which do not have a noun phrase assigned to them. Once the processor fetches 4 bytes from the PTAB it knows exactly which byte is a valid noun phrase. This is because for every valid noun phrase there is a corresponding bit in the BITMAP register which is set to 1.

In this state, the logic controller calculates the distance between each of the four input sentence noun phrases and their corresponding example sentence noun phrases. This involves loading the ADUs and the multipliers in a particular sequence, collecting the intermediate results and synchronizing the operation of all the execution units. The loading operations are performed by 4 concurrent processes which are synchronized by the logic controller. The basic requirement is that the execution

Table 5.1: Operation Scheduling

Time	operation 1	operation 2
0	T1 = GD + PD	T1 = q1 * SD
1	T2 = T1 + ND	
4		T2 = q2 * T2
8	T1 = T1 + T2	T1 = w2 * ASD
9	T2 = T1/(q1+q2)	
12		T2 = w1 * ATD
16	T1 = T1 + T2	
17	T2 = T1/(w1+w2)	

units must be loaded with operands at the same time and must produce results at the same time. All operations are floating point operations. The add operation takes 1 unit of time and the divide and multiply operations take 4 units of time. The sequence in which the operations are executed is shown in table 5.1. The first column denotes the time at which the operations in columns 2 and 3 are started. From the first row in table 5.1 it appears as if the results of operation 1 as well as the results of operation 2 are being assigned to register T1 simultaneously. This is not the case. Only the operations are being scheduled at time 0. The result of operation 1 will be available 1 time unit later in T1. The results of operation 2 are available 4 time units later when its safe to load T1 with any value.

S_5 : In this state the processor adds the distances that it calculated in the previous state to the running sum stored in the register MIN_DIST.

S_6 : The processor has to determine whether there are any more noun phrases that remain to be handled. For this, the BITMAP register is shifted right by 4 bits. If the most significant bit still happens to be 1 then it means that there are still some noun phrases which must be handled. The processor then changes state to S_2.

However, if the last noun phrase has been taken care of then the distance between the input sentence verb phrase and the example sentence verb phrase is calculated.

S_7 : In this state the processor adds the VCD to the running sum stored in the register CUR_DIST.

S_8 : The distance of the current sentence, stored in the register CUR_DIST, is compared with the minimum distance calculated so far which is stored in the register MIN_DIST. If the value in CUR_DIST is less than the value in MIN_DIST then MIN_DIST is loaded with the value in CUR_DIST. The base address of the current sentence is stored in the register MIN_SENTENCE_BASEADDR.

The processor checks if the current sentence is the last sentence in the example base. If it is the last sentence then the processor changes state to S_9. Otherwise the processor sets the CUR_PTAB_BASE, CUR_PTAB_OFFSET, CUR_ASDTAB_BASE, CUR_ASDTAB_OFFSET to point to the next sentence. The BITMAP string is reloaded from memory. The SRC_PTAB_OFFSET and SRC_ASDTAB_OFFSET are reset and state changes to S_2.

S_9 : This state is entered when the processor finishes processing the last sentence in the data set. The minimum distance calculated and the base address of the minimum distance sentence are written into memory at an address pointed to by the register WORK_AREA_ADDRESS. The processor then changes state to IDLE.

IDLE : The processor goes into a loop waiting for the reset signal to be activated again.

5.7.2 The ADU

The ADU has been implemented as a verilog model. It has two ports, *operand_1* and *operand_2* for two 32-bit operands and a *result* port for the result of an operation. The *command* port indicates the operation to be performed, i.e, addition or division.

The *go* port enables the operation and the *done* port signals the completion of an operation.

5.7.3 The Multiplier

The Multiplier has been implemented as a verilog model. *operand_1* and *operand_2* ports hold the two 32-bit operands while *result* holds the 32-bit result of the multiplication. The *go* port enables the operation and the *done* port signals the completion of an operation. The *command* port indicates the function to perform, i.e, multiplication or pass.

5.7.4 The Floating-point Comparator Unit

This unit has two ports *op_1* and *op_2* which hold the two input 32-bit operands. It has a *res* port which outputs the lesser of the two operands. The *enable* port enables the operation and the *done* port signals the completion of the comparison.

5.7.5 Integer Adder Unit

The integer adder is a carry lookahead adder. It has two ports *op_1* and *op_2* which hold the two operands and a *sum* port which holds the sum. The port *c_0* holds the carry in to the zeroth bit and the port *c_out* holds the carry out from the last bit. The *go* port enables the addition operation and the *done* port signals the completion of the operation.

5.7.6 The Data Fetch and Dispatch Unit

In our implementation of the MiDiCal processor we have integrated this unit into the logic controller. This unit implements the bus read and write protocols. It has a 16 bit register, MAR, which drives the 16 bit unidirectional address bus. The 32 bit

register, MDR, holds the data which drives the 32 bit data bus when the processor is writing to memory. The MDR also holds the data that is read in by the processor from memory.

5.7.7 Register Files

Most of the registers are packed into four register files. MiDiCal has three 32-bit floating-point register files (FP-1, FP-2, FP-3) and one 16-bit integer register file(INT-REG).

FP-1 has 6 floating point registers - REG_q1, REG_q2, REG_w1, REG_w2, REG_SD and REG_ASD. It has two read ports *r1* and *r2* and a write port *w1*. Each one of these ports has associated with it an enable port(*en1*, *en2* and *en3*), and an address port(*a1*, *a2* and *a3*).

FP-2 has 6 floating point registers - REG_GD, REG_ND, REG_PD, REG_wsum, REG_qsum and REG_vcd. It has two read ports *r1* and *r2* and a write port *w1*. Each one of these ports has associated with it an enable port, *enr1*, *enr2* and *enw1*, and an address port, *ar1*, *ar2* and *aw1*.

FP-3 has 9 registers - ex1_T1, ex1_T2, ex2_T1, ex2_T2, ex3_T1, ex3_T2, ex4_T1, ex4_T2 and CUR_DIST. It has 12 read ports *r1* to *r12* and 9 write ports *w1* to *w9*. Each write port has an enable port associated with it and an address port associated with it.

INT-REG has thirteen 16-bit registers. It has 2 read ports, *r1* and *r2*, and 1 write port, *w1*. Each port has an enable port and an address port associated with it.

The remaining registers are independent registers in the sense that they have not been pooled into register files.

5.7.8 Noun Phrase Comparator Unit

The MiDiCal processor fetches a 4 noun phrases from the source PTAB and a similar number from the current PTAB. Each noun phrase byte contains the four attributes- SD, GD, PD, ND. The noun phrase comparator unit takes the two words as operands. It checks each one of these attributes and sets some flags if the attributes match.

The interface includes two 32-bit operands *op_1* and *op_2*, an enable pin *en* and 16 output flags.

5.7.9 ASD comparator

The processor fetches 4 bytes from the source ASD table and 4 bytes from the current ASD table. Each byte holds the additional semantic difference for a corresponding noun phrase in the PTAB. The ASD comparator takes these two 32-bit operands and compares them "bytewise". The four output flags F_ASD1_EQ, F_ASD2_EQ, F_ASD3_EQ, F_ASD4_EQ are set if the source ASD word and the current ASD word match in the first, second, third and the fourth bytes respectively.

The interface includes two 32-bit operands *op_1* and *op_2*, an enable pin *en* and 4 output flags F_ASD1_EQ, F_ASD2_EQ, F_ASD3_EQ, F_ASD4_E.

5.8 Simulation Results

This section contains simulation results of the MiDiCal processor. The processor was simulated using a test bench circuit similar to the one used for the simulation of the *DiSc* processor. However, this test bench was written in the Verilog hardware description language. Table 5.1 contains the parameter values used in the simulation.

Table 5.2: MiDiCal Simulation Parameters

BITMAP	q1	q2	q1+q2	w1	w2	w1+w2
2	2.0	0.5	2.5	1.0	1.0	2.0
SD	**GD**	**ND**	**PD**	**ASD**	**VCD**	**W.A.Address**
0.5	0.5	0.5	0.5	0.5	0.5	1024

Table 5.3: Test Data 1

Sentence No.	NP1	NP2	ASD1	ASD2	VERB
Input Sentence	FF	FF	FF	FF	00
Example 1	FF	FE	FF	FF	00
Example 2	0F	FE	8F	FF	11

5.8.1 Test Case 1

The test data is shown in table 5.2. The input sentence contains just two noun phrases. The example base is shown to contain two examples. The processor calculates the distance of the input sentence from the two sentences in the example base and rightly indicates the first sentence in the example base as the sentence closest in meaning to the input sentence.

5.8.2 Test Case 2

Table 5.3 shows the second test data set. The input sentence contains 5 noun phrases. The BITMAP field is thus set to show 5 noun phrases. The example base contains 2 examples. The second example matches the input sentence more closely. The processor thus outputs the second sentence as the result.

Table 5.4: Test Data 2

Sentence	NP1	NP2	NP3	NP4	NP5
Input Sentence	FF	FF	FF	FF	FF
Example 1	0F	FC	FF	FF	FF
Example 2	FF	FE	FF	FF	FF

Sentence	ASD1	ASD2	ASD3	ASD4	ASD5	VERB
Input Sentence	00	00	00	00	00	AA
Example 1	FF	01	00	00	00	0A
Example 2	00	01	00	00	00	AA

5.9 Future Directions

This sections suggests some improvements to the MiDiCal processor. The improvements are architectural enhancements. Logic gate level optimizations are not covered here as the chip was modeled only at a behavioral level.

5.9.1 Bit-sliced Architecture

The MiDiCal processor, as it was implemented, contains only four execution units which can operate on four noun phrases at a time. A better solution would be to build a **bit-sliced** architecture which can scale with the number of noun phrases that must be handled. Those sentences which have more number of noun phrases can have more bit sliced processors working on them.

5.9.2 Choosing 'n' Sentences

The processor was designed to select just one sentence from the example base which is closest in meaning to the input sentence. Often it may be necessary to select more than one sentence from the example base. The processor must be upgraded to incorporate this feature into it.

5.9.3 Pipeline Architecture

The present architecture is a superscalar architecture. All the four execution units operate in parallel. There is a lot of scope to pipeline operations. The memory operations can be pipelined with the minimum distance calculation operations. This would significantly improve performance.

5.9.4 Low Power Design

It is very important to conserve power in a massively parallel processor. So another important improvement is to build a low power version of the MiDiCal processor. The main power consuming units are the multipliers, adders and the dividers. When these units are idle then power can be cut off to them.

5.9.5 Miscellaneous Issues

Fault tolerance issues have not been handled in this implementation. There must be some way to warn the master processor in case the MiDiCal processor is performing faulty calculations.

The processor, in its current implementation, runs to completion once it is reset. It must be modified so that the master processor can interrupt it at any time and then restart it from the point of interruption.

Chapter 6

Conclusions

A massively parallel architecture for the implementation of HEBMT was proposed. Two main bottlenecks in the implementation of HEBMT were identified viz.

- Dictionary search procedure.
- Minimum distance calculation procedure.

ASIC solutions were proposed to overcome the two bottlenecks. The *DiSe* processor was modeled at the behavioral level. This processor was designed to solve the dictionary search problem. The MiDiCal processor was modeled at the behavioral level. This processor was designed to solve the minimum distance calculation problem. The two behavioral models can serve as a base for more detailed(gate level, MOS level, etc.) design. Future directions were suggested for improving the performance of the two ASICs.

Appendix A

DiSe Component Definitions

The chapter describes the components used in the structural modelling of the *DiSe* processor. All the components have been documented in VHDL.

A.1 Multiplexor

```
entity mux is
        port (
             i0 : in bit_vector(7 downto 0);
             i1 : in bit_vector(11 downto 0);
             y : out bit_vector(11 downto 0);
             select : in bit
        );
end mux2;
```

A.2 Transparent Latch

```
entity latch is
```

```
        generic (
                width : positive ;
                Tpd : Time := unit_delay );
        port (
                d : in bit_vector(width-1 downto 0);
                q : out bit_vector(width-1 downto 0);
                en : in  bit
        );
end latch;
```

A.3 Buffer

```
entity buffer is
        generic (
                Tpd : Time unit_delay );
        port (
                a : in bit_8 ;
                b : out bus_bit_8 bus;
                en: in bit ;
        );
end buffer;
```

A.4 Program Counter

```
entity PC is
        port (
                d_in : in bit_12 ;
```

```
            d_out: out bus_bit_12 bus ;
            clear : in  bit ;
            latch_en : in bit ;
            out_en : in bit
        );
end PC;
```

A.5 General Register File

```
entity GEN_REG_FILE is
        port (
            rp_1 : out bus_bit_12 bus ;
            rp_2 : out bus_bit_12 bus ;
            wp_gr: in bit_12 ;
            rad_1: in bit_3 ;
            rad_2: in bit_3 ;
            a3 : in bit_3 ;
            en1,en2,en3 : in bit
        );
end GEN_REG_FILE ;
```

A.6 Source Register File

```
entity SRC_REG_FILE is
        port (
            RD : out bus_bit_8 bus ;
            en1 : in bit ;
```

```
            WR : out bit_8 ;
            en2 : in  bit ;
            next, init : in bit ;
            count_16 : in bit
        );
end SRC_REG_FILE ;
```

A.7 Arithmetic Unit

```
entity ALU is
        port (
            op_1, op_2 : in bit_12 ;
            Result : out bus_bit_12 bus ;
            command : in bit ;
            carry_in_0 : in bit ;
            enable : in bit
        );
end ALU;
```

A.8 Comparator

```
entity Comparator is
        port (
            op_1, op_2 : in bit_12 ;
            enable : in bit ;
            EQ,LT : out bit ;
            GT : out bit
```

```
        );
end Comparator;
```

Appendix B

MiDiCal Component Definitions

This appendix describes the interface definitions of some of the components used in the structural definition of the MiDiCal processor.

B.1 The ADU

```
entity ADU is
        port (
             result : out bit_vector(31 downto 0);
             done : out bit ;
             operand_1 : in bit_vector(31 downto 0);
             operand_2 : in bit_vector(31 downto 0);
             command : in bit;
             go : in bit
        );
end ADU;
```

B.2 The Multiplier

```
entity MULTIPLIER is
        port (
            result : out bit_vector(31 downto 0);
            done : out bit ;
            operand_1 : in bit_vector(31 downto 0);
            operand_2 : in bit_vector(31 downto 0);
            command : in bit;
            go : in bit
        );
end MULTIPLIER;
```

B.3 Floating-point Comparator Unit

```
entity FLOAT_COMPARE is
        port (
            res : out bit_vector(31 downto 0);
            done : out bit ;
            op_1 : in bit_vector(31 downto 0);
            op_2 : in bit_vector(31 downto 0);
            enable : in bit
        );
endFLOAT_COMPARE ;
```

B.4 Integer Adder Unit

```
entity INT_ADDER is
```

```
        port (
            sum : out bit_vector(31 downto 0);
            done : out bit ;
            c_out : out bit ;
            op_1 : in bit_vector(31 downto 0);
            op_2 : in bit_vector(31 downto 0);
            c_0 : in bit ;
            command : in bit;
            go : in bit
        );
end INT_ADDER;
```

B.5 Noun-phrase Comparator Unit

```
entity NP_COMPARATOR is
        port (
            flags : out bit_vector(15 downto 0);
            op_1 : in bit_vector(31 downto 0) ;
            op_2 : in bit_vector(31 downto 0) ;
            en : in bit
        );
end NP_COMPARATOR;
```

B.6 ASD Comparator Unit

```
entity ASD_COMPARATOR is
        port (
```

```
            F_ASD_EQ : out bit_vector(3 downto 0);
            op_1 : in bit_vector(31 downto 0) ;
            op_2 : in bit_vector(31 downto 0) ;
            en : in bit
        );
end ASD_COMPARATOR;
```

B.7 Register File: FP-1

```
entity FP-1 is
        port (
            r1 : out bit_vector(31 downto 0) ;
            r2 : out bit_vector(31 downto 0) ;
            w1 : in bit_vector(31 downto 0) ;
            ar1 : in bit_vector(2 downto 0) ;
            ar2 : in bit_vector(2 downto 0) ;
            aw1 : in bit_vector(2 downto 0) ;
            enr1 : in bit ;
            enr2 : in bit ;
            enw1 : in bit
        );
end FP-1;
```

B.8 Register File: FP-2

```
entity FP-2 is
        port (
```

```
            r1 : out bit_vector(31 downto 0) ;
            r2 : out bit_vector(31 downto 0) ;
            w1 : in bit_vector(31 downto 0) ;
            ar1 : in bit_vector(2 downto 0) ;
            ar2 : in bit_vector(2 downto 0) ;
            aw1 : in bit_vector(2 downto 0) ;
            enr1 : in bit ;
            enr2 : in bit ;
            enw1 : in bit
        );
end FP-2;
```

B.9 Register File: INT-REG

```
entity INT-REG is
        port (
            r1 : out bit_vector(31 downto 0) ;
            r2 : out bit_vector(31 downto 0) ;
            w1 : in bit_vector(31 downto 0) ;
            ar1 : in bit_vector(3 downto 0) ;
            ar2 : in bit_vector(3 downto 0) ;
            aw1 : in bit_vector(3 downto 0) ;
            enr1 : in bit ;
            enr2 : in bit ;
            enw1 : in bit
        );
end INT-REG;
```

References

[1] Renu Jain, R.M.K. Sinha, and Ajai Jain, 1995, "Pattern Directed Hybrid Approach to Machine Translation Through Examples",Proceedings of Symposium on Natural Language Processing '95(SNLP'95).

[2] H. Kitano, "Challenges Of Massive Parallelism".

[3] Abhaya Astahana, Mark Cravats, Paul krzyzanowski, 1994, "An Experimental Active memory Based I/o", Computer Architecture News, Vol 22, No. 4, September 1994.

[4] Harold S. Stone, 1992, "Optimal Partitioning Of Cache Memory", IEEE Transactions On Computers, Vol. 41, No.9, September 1992.

[5] H.Kitano, 1993a, "A Comprehensive And Practical Model Of Memory Based Machine Translation", Proceedings Of IJCAI-93.

[6] E.Sumita, K.Oi, H.Iida, T.Hiyuchi, N.Takahashi, H.Kitano, 1993b, "Example-Based Machine Translation On Massively Parallel Processors", Proceedings Of IJCAI-1993.

[7] Stenstrom Per, Lund University, 1998, "Reducing Contention In Shared Memory Multiprocessors", IEEE Computer.

[8] Lewis W.Tucker, George G. Robertson, 1998, "Architecture And Applications Of The Thinking Machines", IEEE Computer.

[9] O.Furuse and H.Iida, 1992, "Cooperation between Transfer and Analysis in Example-Based Framework", Proceedings of COLING-92.

[10] J.Hutchins, 1993, "Latest Developments in Machine Translation Technology: Beginning a new era in MT Research", MT Summit IV, Japan.

[11] M.Nagao, 1985, "A Framework of a Mechanical Translation Between Japanese and English by Analogy Principle", Elithorn, A.Banaji, R.(eds:) Artificial And Human Intelligence, Amsterdam: North Holland.

[12] C.Reisbeck, R.Schank, 1990, "Inside Case-Based Reasoning", Lawrence Erlbaum Associates.

[13] S.Sato, M.Nagao, 1990, "Towards Memory-Based Translation", Proceedings of COLING-90.

[14] R.M.K.Sinha and Sivaraman, 1993, "ANGLA-BHARATI: A Machine Aided Translation System from English to Indian Languages - An Overview", Technical Report TRCS-93-173, Department of Computer Science and Engineering, IIT Kanpur.

[15] C.Stanfill, D.Waltz, 1986, "Towards Memory-Based Reasoning", Communications of the ACM, vol. 29, No. 12.

[16] Hamacher, Vranesic, and Zaky, 1990, Computer Organization,McGraw Hill International Edition", Singapore.

[17] Lipsett R, Schaefer F.C, Ussery C, 1988, VHDL: Hardware Description and Design, Boston, Kluwer Academic.

[18] Bhasker J, A VHDL Primer, 1988, Engelwood Cliffs, NJ, Prentice Hall.

[19] Pradhan D.K, Fault-Tolerant Computer System Design, 1996, Prentice Hall.